An Empirical Study on Remote Code Execution in Machine Learning Model Hosting Ecosystems

Anonymous Author(s)

Abstract

Model-sharing platforms (*e.g.*, Hugging Face) have become central to modern ML development, enabling easy sharing and reuse of pre-trained models. However, this flexibility introduces serious risks, as models may execute untrusted code during loading (*e.g.*, via `trust_remote_code`). This paper presents the first large-scale empirical study of custom model-loading practices across five major platforms, examining their prevalence, risks, and developer perceptions. We quantify how often models rely on custom code and identify those executing arbitrary Python files. Using Bandit, CodeQL, and Semgrep, we detect security smells and categorize findings by CWE identifiers, complemented by YARA-based malware signature scans. We further analyze each platform's documentation, APIs, and safeguards, and qualitatively study over 600 community discussions. Our results reveal widespread unsafe defaults, inconsistent security enforcement, and pervasive developer confusion about the risks of remote code execution.

CCS Concepts

• **Security and privacy** → **Systems security**; **Software security engineering**; • **Software and its engineering** → *Software infrastructure*.

Keywords

Large Language Models (LLMs), Software Security, Remote Code Execution, Model Hub

ACM Reference Format:

Anonymous Author(s). 2026. An Empirical Study on Remote Code Execution in Machine Learning Model Hosting Ecosystems. In *Proceedings of The Mining Software Repositories (MSR), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large Language Models (LLMs) have been increasingly used in day-to-day conversation and assisting tasks [3, 43, 53]. These models are based on different transformer architectures [68]and their advancements. These have enabled the creation of models with unprecedented scale, often comprising billions or even trillions of parameters [43]. Models are continuously reused, re-tuned, and evaluated for new tasks. **Model hubs** (or model registries) like Hugging Face play a critical role in this ecosystem by providing a centralized platform for hosting and sharing pre-trained models and datasets [72]. As of October 2025, Hugging Face alone hosts around 1.7 million models, fostering an open and collaborative environment for developers and researchers worldwide.

While model hubs and their supporting libraries (*e.g.*, `transformers` [70] and PyTorch [61]), enable the seamless distribution of model weights, some models inherently require code execution to function correctly [24]. Early neural network architectures relied on standardized, composable layers that could be fully described through configuration files. In contrast, current LLMs often introduce non-standard architectural components (*e.g.*, custom attention mechanisms, domain-specific preprocessing steps, and hardware-aware optimizations) that cannot be easily serialized without accompanying executable code [71]. For example, when researchers develop a new transformer variant with a novel positional encoding scheme, they must distribute not only the trained weights but also the accompanying Python code that specifies how those weights are applied during inference. Without this code, downstream users would need to manually re-implement the architecture, making model sharing inefficient and, in many cases, impractical [15].

Allowing code to run during model loading increases flexibility but also introduces security risks [72]. One security issue arises from unsafe serialization formats such as Python's `pickle`, which can execute arbitrary code during deserialization via the `__reduce__` method—turning model files into potential attack payloads [4, 32]. Prior work has shown that malicious pickle-based models in the wild have been used to deploy reverse shells and steal credentials [5, 72]. Another distinct security problem comes from *custom remote code execution*, enabled when users load models with flags such as `trust_remote_code=True`. This allows arbitrary Python modules provided by model authors to run locally, extending the attack surface beyond serialized data to unverified source code.

Consequently, loading models from public hubs creates implicit trust relationships among users, model authors, and platforms (a trust that is often misplaced [30]). While previous work has examined deserialization attacks [20, 20], ***the prevalence and risks of custom remote code execution during model loading remain largely unexplored***. Therefore, this paper closes this gap through a large-scale empirical study across five major model-sharing platforms (Hugging Face [25], OpenCSG [48], ModelScope [40], OpenMMLab [52], and PyTorch Hub [61]). We first quantify how often models rely on custom loading code and identify those that execute arbitrary Python files. We then apply multiple static analysis tools (Bandit, CodeQL, and Semgrep) to detect potential vulnerabilities and categorize them by CWE identifiers. In parallel, we analyze each platform's documentation, APIs, and security controls to assess mitigation practices, nd qualitatively examine over 600 developer discussions from GitHub, Hugging Face, PyTorch Hub, and Stack Overflow to capture community perceptions and misconceptions about security and usability.

The contributions of this work[1] are:

- The first cross-platform, large-scale measurement study of untrusted model code execution across five major model-sharing platforms: Hugging Face [25], ModelScope [40], OpenCSG [48], OpenMMLab [52], and PyTorch Hub [61].
- We systematically detect and categorize security weaknesses using three static analyzers in around 45,000 repositories containing custom code. Moreover, we incorporate signature-based malicious pattern detection using YARA [35] to identify potential payloads.
- We analyze platform-level defenses, including warning systems, static and dynamic scanning, and trust flag mechanisms.
- We create a taxonomy about developers' perception about remote code execution during model loading after examining around 600 developer discussions from forums, GitHub issues, pull requests, and Q&A sites.

2 Background

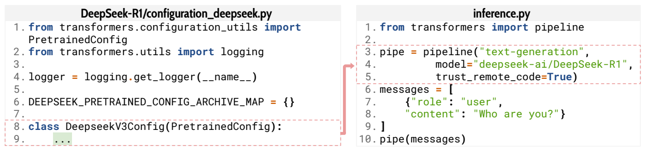
2.1 Model Loading with Executable Code

Unlike traditional data files, modern ML models may require code execution for technical reasons. Early neural networks consisted of standardized layers that could be described solely by configuration, but contemporary architectures implement novel mechanisms (*e.g.*, custom attention patterns, domain-specific preprocessing, *etc.*) that cannot be expressed without executable code [71]. Thus, when developers implement a new transformer variant with unique positional encoding, they must ship *both* the trained weights and the Python code defining how those weights interact. The alternative would require every user to manually reconstruct the architecture, making large-scale model sharing impractical [15].

This technical need manifests through platform APIs that are simple to use. When developers call `from_pretrained` or `pipeline` methods, the `transformers` library downloads multiple files, including Python modules that execute with full system privileges when remote code trust flags are enabled. Whenever the repository contains specific entry-point files, such as `modeling_*.py`, `tokenizer.py`, or `hubconf.py`, they are automatically imported and executed as part of the model initialization process. While these files often contain legitimate code that defines how the model operates, an attacker can embed malicious payloads in them that would execute with the same privileges as any other local Python process. This means that enabling `trust_remote_code` (for `transformers`) or `trust_repo` (for PyTorch Hub) effectively grants remote repositories the ability to run arbitrary Python code on the host machine. Platforms like Hugging Face host over 1.7 million models, with thousands added daily, making manual review impractical and automated scanning insufficient [34].

[1]This study's replication package is available at [1].

To illustrate, Figure 1 shows how model loading can trigger code execution. On the left, a developer defines a custom configuration class (`DeepseekV3Config`) that extends the `transformers` library's base configuration. This file, stored in the model repository, contains Python code that can be executed when the model is loaded. On the right, a user loads this same model from Hugging Face using the `pipeline` API and sets `trust_remote_code=True`. This flag tells the library to trust and run any Python code provided by the remote repository, effectively downloading and executing unverified scripts from the internet.



DeepSeek-R1/configuration_deepseek.py

```python
from transformers.configuration_utils import
PretrainedConfig
from transformers.utils import logging

logger = logging.get_logger(__name__)

DEEPSEEK_PRETRAINED_CONFIG_ARCHIVE_MAP = {}

class DeepseekV3Config(PretrainedConfig):
    ...
```

inference.py

```python
from transformers import pipeline

pipe = pipeline("text-generation",
        model="deepseek-ai/DeepSeek-R1",
        trust_remote_code=True)
messages = [
    {"role": "user",
    "content": "Who are you?"}
]
pipe(messages)
```

Figure 1: Example of a custom configuration script (left) and loading a custom model (right).

This customization mechanism introduces a serious security risk: an attacker can upload a model repository containing **malicious code** to deliver payloads like reverse shells, keyloggers, or data exfiltration scripts [72]. Since the customization code executes automatically during model loading, users who enable `trust_remote_code` may unknowingly grant full system access to untrusted code. This behavior makes large-scale model sharing both powerful and potentially dangerous, especially when combined with the high volume of new models uploaded daily.

2.2 Code Smells & Security Smells

Code smells are indicators of poor design or implementation choices that may not immediately cause failures but often lead to maintainability issues and increased defect risk [10, 54]. They typically reflect violations of good design principles, making software harder to evolve and more error-prone.

A specific subset of code smells, called ***security smells***, is associated with patterns that may introduce or signal the presence of vulnerabilities [11, 63, 64]. These patterns do not always constitute exploitable vulnerabilities but highlight code areas where security controls are weak or outdated.

2.3 Threat Model

Custom model loading introduces complex trust boundaries among three main stakeholders: *model creators*, *platform maintainers*, and *model consumers*. Figure 2 provides an overview of our threat model, which examines how these boundaries can be exploited when platforms allow arbitrary code execution during model loading.

Adversary Assumptions. We assume that attackers have accounts on model platforms and can upload models or modify existing repositories. Adversaries may also leverage community features, such as discussions or pull requests, to disseminate or promote malicious content. We consider two adversary types: (1) a *malicious developer* who intentionally uploads a model with harmful customization

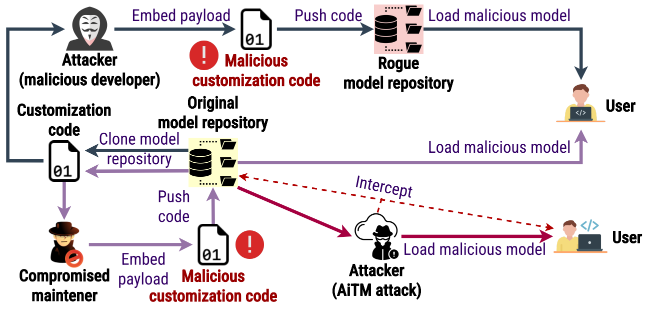


Figure 2: Threat model overview

code, and (2) a *compromised maintainer* whose credentials or tokens were hijacked to distribute poisoned models under the guise of a reputable author. Importantly, not all threats arise from deliberate attacks. Model contributors who are unaware of secure publishing practices may unintentionally introduce insecure code, effectively expanding the threat surface without adversarial intent.

Threat Scenarios. Three threat scenarios exist (**S1**–**S3**):

S1 **Malicious Fork.** An attacker downloads an existing benign model M, changes the custom code with a harmful payload (*e.g.*, reverse shell), and uploads a modified model M' to their own (rogue) repository while presenting it as an enhanced or compatible version of the original. Model consumers would need to set `trust_remote_code=True` to be able to use the model, which would lead to the execution of the malicious payload.

S2 **Compromised Trusted Account.** A trusted maintainer account is hijacked, allowing the attacker to upload a modified model M' with malicious accompanying custom code directly to a legitimate, widely used repository. Consumers may trust the model due to its reputation, verified badges, or high download counts. Similar to Scenario 1, attackers can embed arbitrary Python logic in initialization files or leverage dependency manipulation. Social engineering and trust hijacking further increase the likelihood of exploitation, as users are more inclined to enable `trust_remote_code=True` for "trusted" sources.

S3 **Attacker-in-the-middle (AiTM).** An attacker intercepts or tampers with the model distribution channel and modifies the model's custom code during transfer or dependency resolution (*e.g.*, via compromised mirrors, registries, or proxy layers). This can also occur indirectly through poisoning or replacing cached artifacts stored by hosting platforms (*e.g.*, Hugging Face cache directories), allowing the attacker's modified version to be loaded even if the original upstream repository remains clean. As a result, when users enable `trust_remote_code=True`, they may execute the injected payload from the cached or intercepted model, effectively transforming a previously benign artifact into a malicious one. Attackers may exploit dependencies and supply-chain manipulation, injecting malicious payloads at download or cache resolution time. Since the model is cached locally, future loads may execute the attacker's payload even without further network interaction.

Not all risks stem from malicious actors. Model creators with limited security awareness may unintentionally include unsafe initialization code, hard-coded credentials, or insecure dependency calls. Although unintentional, such models can still be weaponized post-deployment, expanding the platform's overall attack surface without deliberate adversarial behavior. Weak sandboxing, overly permissive dependencies, and a lack of static or runtime checks allow insecure code to run automatically during model loading, making these models soft targets for downstream exploitation.

Trust Relationships. Model consumers implicitly trust platform interfaces and configuration defaults (*e.g.*, `trust_remote_code=True` or `trust_repo=True`) to safely retrieve and execute model code. This trust is often amplified by perceived platform reputation or download counts, which may lead users to overlook warning prompts or disable security mechanisms for convenience. Platform maintainers, in turn, trust model contributors to follow safe publishing practices, while contributors depend on the platform to enforce isolation and verification mechanisms. The intersection of these assumptions creates a vulnerable trust boundary.

3 Methodology

3.1 Research Questions (RQs)

As shown in Figure 3, we answer four RQs that explore the *prevalence*, *risks*, and *developers' perceptions* of custom model loading across model hubs.

RQ1: To what extent is custom model loading required?

Model-sharing platforms, such as Hugging Face and ModelScope, enable developers to provide custom code for loading or configuring models. While these capabilities increase flexibility and support novel architectures, they also introduce security risks within the ecosystem. In this RQ, we investigate how many models hosted on these platforms include custom code that is required to be executed upon model loading.

RQ2: Do remote code implementations for custom models contain vulnerabilities, security smells, or malicious payloads?

Allowing arbitrary code execution raises concerns about the introduction of vulnerabilities and insecure practices. Previous work has shown that code provided by the community may contain security smells, such as unsafe deserialization [4, 5, 72]. In this RQ, we perform a systematic analysis of the models' customization code to determine the prevalence of vulnerabilities and potential exploit vectors.

RQ3: What do the platforms offer for developers to mitigate the execution during model loading?

Platform hubs play a crucial role in enforcing safe defaults, providing comprehensive documentation, and implementing technical safeguards (*e.g.*, sandboxing, warning banners, or permission systems). In this RQ, we investigate the existing security mechanisms provided by various platforms. We examine whether platforms provide static or dynamic checks, whether they expose developers and end-users to explicit warnings when running custom code, and how policies such as `trust_remote_code` or `trust_repo` flags or isolated execution environments are enforced in practice.

RQ4: What are the developers' concerns around code execution during model loading?

Beyond technical vulnerabilities, it is crucial to understand the perspective of developers who contribute to and use these models. Their concerns may range from usability (*e.g.*, friction in using security mechanisms) to trustworthiness (*e.g.*, fear of executing malicious code) and maintainability (*e.g.*, lack of long-term platform support for their contributions). In this RQ, we collect and analyze developer discussions from Hugging Face and PyTorch Hub forums, GitHub discussions, pull requests, issues, and StackOverflow Q&A platforms to understand practitioners' concerns, misconceptions, and expectations.

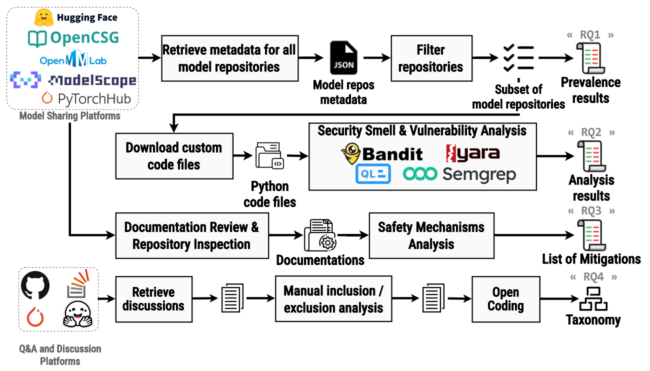


Figure 3: Methodology Overview

3.2 Platform Selection

We first examined a recent list of popular model-sharing platforms compiled by Jian *et al.* [72], which identified the top 15 model hubs. We then applied the following inclusion criteria to determine whether a given model hub (henceforth referred to as a "platform") qualified for our study. A platform was included only if it satisfied *all* of the following conditions: (1) it is publicly accessible; (2) it provides practical means to retrieve information from all hosted model repositories (*e.g.*, through APIs or web crawling); (3) the hosted models can be programmatically fetched and instantiated through standard model-loading APIs provided by `transformers`, `torch.hub`, or their respective wrappers, without requiring any reimplementation; and (4) it supports libraries or mechanisms that permit the execution of custom code during model loading. After this manual analysis, we identified **5** platforms for inclusion in our study: Hugging Face [25], OpenCSG [48], ModelScope [40], OpenMMLab [52], and PyTorch Hub [61].

3.3 Model Repositories Selection

After selecting the platforms, we collected metadata for all model repositories published within these platforms. Specifically, we captured the repository's access URLs, tags, and file list. Next, we applied the following inclusion criteria to identify model repositories that require custom code execution during model loading. A repository was included if it satisfied any of the following criteria: **(i)** it was tagged with the `custom_code` tag; or **(ii)** it contained one of the files `tokenizer.py`, `__init__.py`, or `hubconf.py`; or **(iii)** it included a Python file whose name began with `modeling_`, `tokenization_`, or `configuration_`. These conditions are based on the documentation of the `Pytorch Hub` [60] and the `transformers` library [24]. The OpenMMLab platform, on the other hand, has its own library for loading custom models. Therefore, we included all the repositories listed on this platform in our analysis.

3.4 Security Smell, Vulnerability, and Malicious Payload Analysis

After collecting the models, we used three static analyzers to identify security smells and potential vulnerabilities: Bandit [56], CodeQL [13], and Semgrep [62]. To further identify malicious patterns and payload signatures, we employed YARA [35]. These tools provide complementary coverage, combining lightweight static analysis with signature-based detection of malicious code.

Bandit (v1.8.6). It is a security linter that statically inspects the abstract syntax tree (AST) of Python code to detect common vulnerabilities such as the use of unsafe functions (`eval`, `exec`, `pickle.load`), weak cryptographic algorithms, hardcoded credentials, and insecure temporary file creation. It also maps findings to *Common Weakness Enumeration (CWE)*, which is a list of common types of software vulnerabilities [36]. We executed Bandit recursively on all Python files extracted from each model repository, generating structured JSON outputs for aggregation and comparison.

CodeQL (v2.15.0). It performs static analysis by compiling source code into a relational database of program elements (*e.g.*, functions, variables, control flow, and data flow) and executing declarative `QL` queries to detect security flaws. It enables inter-procedural and dataflow analysis for complex vulnerabilities such as injection, path traversal, and insecure deserialization. We executed CodeQL with the official query pack provided by GitHub Security Lab [12].

Semgrep (v1.139.0). It is a lightweight, multi-language static analyzer that uses rule-based pattern matching to detect both general and domain-specific security issues. Unlike CodeQL, which requires query compilation, Semgrep matches syntactic and semantic patterns directly in the codebase, making it efficient for large-scale scanning. We used its built-in rulesets to identify security misconfigurations, unsafe API usage, and insecure imports across various model repositories. Findings were grouped by CWE and severity level to facilitate cross-tool comparison.

YARA (v4.5.2). It is a rule-based pattern-matching engine widely used in malware detection and digital forensics [41, 42]. Unlike the previous static analyzers, which focus on identifying insecure coding patterns or API misuse, YARA detects known malicious behaviors through signature-based matching of strings, byte sequences, and regular expressions. This approach allows it to uncover embedded payloads that may not manifest as conventional security smells—for example, reverse shells, obfuscated network beacons, or credential-stealing scripts. The official YARA GitHub repository [69] provides a reference to a curated list of well-known rule sources [28]. From these 70 publicly available sources, we successfully compiled 7,657 rules from 25 sources to scan the collected repositories (they failed to compile rules due to them being outdated with respect to the version we have used in our work).

3.5 Platform Mitigation

To answer RQ3, we systematically examined each platform to identify mechanisms mitigating unsafe model loading. We first reviewed official documentation and API references to understand declared security policies for custom code execution. We then analyzed open-source repositories (if available) to verify enforcement of safety configurations such as `trust_remote_code` or `trust_repo`. From these sources, we identified their used safeguards (*e.g.*, *static* and *dynamic* checks, warning prompts, sandboxed execution, *etc.*) and compared them across platforms to evaluate consistency, enforcement, and gaps.

3.6 Developers' Concerns

To answer our RQ4, we focused on the Hugging Face and PyTorch Hub forums, GitHub discussions, pull requests, issues, and the Stack Overflow Q&A platform. We searched these platforms using the keywords `trust_remote_code` and `trust_repo`. We identified a total of **418** Hugging Face forum posts, **354** GitHub discussion posts, and **289** Stack Overflow posts. For GitHub issues, there were more than 13,000 retrieved issues, and for GitHub pull requests, more than 4,000 PRs, using the aforementioned query. Thus, we kept only issues and PRs that included any of the search query keywords in their titles. Then, two authors manually filter them in parallel to include them in our study based on their relevance. The Cohen's kappa score is 0.50, indicating a "moderate" level of agreement between the two authors [7]. A senior author with over 10 years of experience resolved the discrepancies.

After manually filtering for relevance to our study, we retained 27 GitHub discussions, 222 GitHub issues, 297 GitHub pull requests, 27 Stack Overflow posts, and 39 Hugging Face discussions. An entry was deemed as relevant if it explicitly discussed the functionality, security implications, integration issues, maintenance actions, or community understanding related to `trust_remote_code` or `trust_repo`, rather than merely mentioning the keyword in a code snippet as shown in the example in Listing 1. It is important to note that although we collected 23 discussions from PyTorch Hub's forums, none of them were relevant to our study.

We then applied an *open coding* approach to the selected posts [8], carefully reading, analyzing, and annotating each post with conceptual labels (*codes*) reflecting developers' expressed sentiments, challenges, and misunderstandings. The coding was conducted collaboratively by the authors, whose software development experience ranged from 4 to 12 years. To ensure consistency, disagreements were discussed in weekly calibration meetings. If there was a discrepancy, the senior author mitigated it. We iteratively reviewed and refined the codes through regular calibration meetings until conceptual saturation was achieved.

4 Results

4.1 RQ1: Prevalence of Custom Models

Table 1 provides a summary of the collected model repositories containing custom code. For Hugging Face, OpenCSG, and ModelScope, approximately 2% to 4% of the models include custom code. In contrast, for OpenMMLab and PyTorch Hub, all available repositories rely on custom code.

Table 1: Result of collected repositories with custom codes.

Platform	# Repos	# Repos with custom code	(%)	Supported Libs.
Hugging Face	1,699,968	35,953	2.11%	`transformers`
OpenCSG	192,556	6165	3.20%	`transformers`
ModelScope	68,736	3193	4.65%	`PyTorch & transformers`
OpenMMLab	16	16	100.00%	`MMengine & PyTorch`
PyTorch Hub	26	26	100.00%	`PyTorch`

Figure 4 depicts the top 10 tag distributions of custom models across Hugging Face, ModelScope, OpenCSG, OpenMMLab, and PyTorch Hub platforms. For Hugging Face, the majority of custom models are used for *text generation* (66.5%), followed by *feature extraction* (approximately 10%). In contrast, for OpenCSG, around 21.8% of the custom models are used for text generation, while roughly 15.1% are associated with *speech-related* tasks. Finally, for ModelScope, the distribution indicates that most custom models are concentrated in *scientific and domain-specific applications*, reflecting its distinct usage patterns compared to the other two platforms. For PyTorch Hub, most models focus on vision, speech, and audio tasks, whereas in OpenMM Lab, model repositories mainly contain custom models for object detection and other vision tasks.

4.2 RQ2: Security Analysis

Bandit Results. Table 2 presents the distribution of vulnerability types (CWE IDs) and the top Bandit issues by severity across five major model-sharing platforms. Across all platforms, **CWE-703 (Improper Check or Handling of Exceptional Conditions)** is by far the most common weakness, especially on Hugging Face (78%) and ModelScope (80.95%), indicating that model repositories frequently include weak or missing exception handling logic. This pattern is associated with the Bandit low-severity rule **B101 (assert statements)**, which alone accounts for 60–80% of all detected issues on most platforms. While these may not directly expose models to immediate exploits, they represent fragile or insecure coding practices that can lead to reliability and maintainability problems.

For Hugging Face, out of 35,953 repositories with custom codes, 3,743 (10.41%) exhibited at least one security smell. In addition to CWE-703, CWE-494 (Download of Code Without Integrity Check) and CWE-259 (Use of Hard-coded Password) were common. For OpenCSG, despite a relatively modest "smelly" repository rate of 2.34%, the absolute number of issues is the highest among all platforms. Unsafe downloads (B615, 15.11%) and unsafe PyTorch loads (B614, 5.81%) were particularly common. ModelScope shows the lowest proportion of affected repositories (0.33%), yet a similar CWE profile dominated by CWE-703 and CWE-259. The high prevalence of B101 issues indicates poor exception-handling practices.

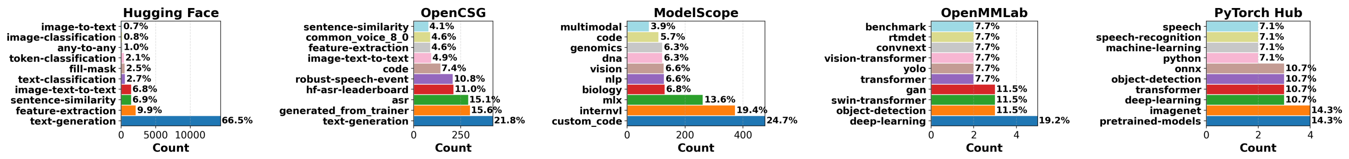


Figure 4: Tag distributions of the custom models across different platforms.

Table 2: Top 3 CWE and top 1 Bandit issue per severity across platforms.

Platform	# Repos Analyzed	# Smelly Repo (%)	Top 3 CWEs	Top 1 Issue per Severity (H, M, L)
Hugging Face	35,953	3,743 (10.41%)	CWE-703 (Exception Handling): 160,967 (78.09%) CWE-494 (No Integrity Check): 19,840 (9.62%) CWE-259 (Hard-coded Password): 13,919 (6.75%)	**H1**. B605 – Starting a process with a shell. (394; 0.19%) **M1**. B615 – Unsafe Hugging Face Hub download. (17,042; 8.01%) **L1**. B101 – Use of assert detected. (157,788; 74.18%)
OpenCSG	192,556	4,503 (2.34%)	CWE-703 (Exception Handling): 40,433 (64.55%) CWE-494 (No Integrity Check): 13,298 (21.23%) CWE-502 (Unsafe Deserialization): 4,426 (7.07%)	**H1**. B605 – Starting a process with a shell. (221; 0.34%) **M1**. B615 – Unsafe Hugging Face Hub download. (9,957; 15.11%) **L1**. B101 – Use of assert detected. (39,717; 60.28%)
ModelScope	68,736	229 (0.33%)	CWE-703 (Exception Handling): 20,490 (80.95%) CWE-259 (Hard-coded Password): 2,584 (10.21%) CWE-494 (No Integrity Check): 1,455 (5.75%)	**H1**. B605 – Starting a process with a shell. (22; 0.09%) **M1**. B615 – Unsafe Hugging Face Hub download. (1,278; 4.95%) **L1**. B101 – Use of assert detected. (20,169; 78.17%)
OpenMMLab	16	12 (75.00%)	CWE-703 (Exception Handling): 848 (86.44%) CWE-259 (Hard-coded Password): 124 (12.64%) CWE-502 (Unsafe Deserialization): 6 (0.61%)	**H1**. B301 – `pickle` used to deserialize untrusted data. (4; 0.41%) **L1**. B101 – Use of assert detected. (848; 86.44%)
PyTorch Hub	26	12 (46.15%)	CWE-78 (OS Command Injection): 42 (35.90%) CWE-703 (Exception Handling): 40 (34.19%) CWE-502 (Unsafe Deserialization): 20 (17.09%)	**H1**. B605 – Starting a process with a shell. (2; 7.69%) **M1**. B301 – `pickle` used to deserialize untrusted data. (1; 3.85%) **L1**. B101 – Use of assert detected. (34; 30.77%)

Although the total number of repositories is small (16) for OpenMMLab, 75% of them contained security smells. For this platform, CWE-703 accounted for 86.44% of issues, and unsafe deserialization (B301) and `eval` usage (B307) were notable medium-severity findings. For PyTorch Hub, nearly half of the repositories (46.15%) contained security issues. Unlike other platforms, CWE-78 (OS Command Injection) was the most common weakness, reflecting the frequent use of shell commands and `eval` functions. This confirms that this platform is particularly susceptible to remote code execution risks.

Semgrep Results. Table 3 summarizes the distribution of vulnerability types (CWE IDs), top OWASP categories, and the most frequent Semgrep rule violations across major model-sharing platforms. Unlike Bandit's results, which primarily had lower-severity coding smells, the Semgrep analysis reveals a *clear concentration of security-critical issues*, more specifically related to unsafe deserialization, code injection, and integrity failures. Across all platforms, **CWE-502 (Deserialization of Untrusted Data)** is the most common weakness, consistently appearing in 50–80% of the flagged repositories. Additionally, CWE-95 (Eval Injection), CWE-676 (Use of Potentially Dangerous Function), and CWE-706 (Improper Handling of Variadic Functions) appear across platforms. The top OWASP categories identified through Semgrep closely align with injection-based threats. **Injection** vulnerabilities dominate on most platforms (*e.g.*, 54.6% on Hugging Face and 67.9% on PyTorch Hub), followed by **Insecure Deserialization** and **Integrity Failures**.

Hugging Face had 8.65% of its repositories affected, with more particularly CWE-502 and CWE-95. A small proportion (0.02%) of repos from OpenCSG showed issues, with CWE-502 dominating (82.76%). ModelScope shows a similar low impact (0.05%), but CWE-502 is still dominant (62.5%). The presence of CWE-22 and CWE-502 highlights insecure file operations and deserialization risks. For PyTorch Hub, there is a highest relative impact (53.85%), with CWE-502 and CWE-95 dominating.

CodeQL Results. Table 4 shows the distribution of CWE IDs and the most frequent CodeQL rule violations among Hugging Face, OpenCSG, and ModelScope as it did not find any issues in OpenMMLab and PyTorch Hub. Across all platforms, **CWE-117 (Improper Output Neutralization for Logs)** and **CWE-020 (Improper Input Validation)** are the most common issues. Additional findings include CWE-079 (Cross-site Scripting), CWE-209 (Information Exposure Through an Error Message), and CWE-022 (Path Traversal) are also frequent. Most issues detected by CodeQL are under the **"Timing attack against secret"** query, accounting for more than 90% of all detections across platforms. This pattern indicates that many repositories rely on non-constant-time cryptographic comparisons, which are vulnerable to side-channel attacks.

About 5.87% of repositories were flagged with CodeQL alerts for Hugging Face. CWE-117 accounted for over half of all detected weaknesses (53.49%), showing poor log sanitization practices. Although the absolute number of flagged repositories is low (32) for OpenCSG, CWE-215 (Information Exposure Through Debug Information) and CWE-730 (OWASP Top Ten 2004 Category A9) stand out. A small proportion of repositories (0.25%) from ModelScope contain CWE-020 (78.26%), CWE-116 (Improper Encoding or Escaping), and CWE-079. Timing-attack patterns again dominate (98.75%), often coupled with weak cryptographic configuration.

YARA Results. Table 5 presents the distribution of YARA signature matches across the model-sharing platforms. The results reveal that while *malware-related signatures are concentrated in a relatively small fraction of repositories*, and across all platforms, the most frequently triggered YARA signatures belong to the category of **environmental evasion indicators** (*e.g.*, Qemu, VMWare, VBox detections).

Among 35,953 repositories of Hugging Face, 2,924 (8.13%) exhibited at least one YARA malicious payload match. The top detections are **JT 3D Visualization format** (57.72%), followed by **VBox**, **Qemu**,

Table 3: Top 3 CWE, OWASP, and Semgrep issues across platforms.

Platform	# Smelly Repos (%)	Top 3 CWEs	Top 3 OWASP	Top 3 Rules
Hugging Face	3,110 (8.65%)	CWE-502 (Unsafe Deserialization): 7,904 (74.54%) CWE-95 (Eval Injection): 1,593 (15.02%) CWE-676 (Dangerous Function): 466 (4.39%)	Injection: 1,894 (50.23%) Integrity Failures: 597 (15.83%) Insecure Deserialization: 595 (15.78%)	pickles in pytorch: 7,252 (56.51%) numpy in pytorch: 3,048 (23.75%) eval detected: 1,502 (11.70%)
OpenCSG	1,141 (0.59%)	CWE-502 (Unsafe Deserialization): 5,449 (82.81%) CWE-676 (Dangerous Function): 451 (6.85%) CWE-95 (Eval Injection): 344 (5.23%)	Injection: 616 (30.36%) Integrity Failures: 493 (24.30%) Insecure Deserialization: 416 (20.50%)	pickles in pytorch: 4,923 (67.73%) numpy in pytorch: 1,279 (17.60%) automatic memory pinning: 414 (5.70%)
ModelScope	581 (0.85%)	CWE-502 (Unsafe Deserialization): 526 (72.65%) CWE-95 (Eval Injection): 144 (19.89%) CWE-706 (Incorrectly-Resolved Name): 28 (3.87%)	Injection: 159 (58.46%) Insecure Deserialization: 38 (13.97%) Integrity Failures: 38 (13.97%)	pickles in pytorch: 484 (46.45%) numpy in pytorch: 351 (33.69%) eval detected: 143 (13.72%)
OpenMMLab	2 (12.50%)	CWE-502 (Unsafe Deserialization): 8 (61.54%) CWE-95 (Eval Injection): 3 (23.08%) CWE-706 (Incorrectly-Resolved Name): 2 (15.38%)	Insecure Deserialization: 8 (38.10%) Integrity Failures: 8 (38.10%) Injection: 3 (14.29%)	avoid pickle: 8 (61.54%) eval detected: 2 (15.38%) non-literal import: 2 (15.38%)
PyTorch Hub	10 (38.46%)	CWE-502 (Unsafe Deserialization): 25 (49.02%) CWE-95 (Eval Injection): 16 (31.37%) CWE-676 (Dangerous Function): 5 (9.80%)	Injection (A03:21): 19 (67.86%) Injection (A01:17): 3 (10.71%) Insecure Deserialization: 2 (7.14%)	pickles in pytorch: 23 (46.94%) eval detected: 16 (32.65%) automatic memory pinning: 5 (10.20%)

Table 4: Top 3 CWE and CodeQL issues across platforms.

Platform	# Smelly Repo (%)	Top 3 CWEs	Top 3 Rules
Hugging Face	2,111 (5.87%)	CWE-117 (Log Injection): 376 (53.49%) CWE-20 (Input Validation): 101 (14.37%) CWE-79 (XSS): 98 (13.94%)	Timing attack against secret: 53,432 (97.82%) Log Injection: 376 (0.69%) All Cryptographic Algorithms: 345 (0.63%)
OpenCSG	32 (0.00%)	CWE-215 (Debug Info Exposure): 22 (23.91%) CWE-730 (ReDoS): 22 (23.91%) CWE-79 (XSS): 17 (18.48%)	Timing attack against secret: 6,931 (94.63%) All Cryptographic Algorithms: 140 (1.91%) Hash Algorithms: 140 (1.91%)
ModelScope	169 (0.25%)	CWE-20 (Input Validation): 18 (78.26%) CWE-116 (Output Encoding): 3 (13.04%) CWE-79 (XSS): 2 (8.70%)	Timing attack against secret: 6,571 (98.75%) All Cryptographic Algorithms: 23 (0.35%) Hash Algorithms: 23 (0.35%)

Table 5: Top 3 YARA issues across platforms.

Platform	# Smelly Repo (%)	Top 3 Rules
Hugging Face	2,924 (8.13%)	JT 3D Visualization format: 32,549 (57.72%) VBox Detection: 7,560 (13.41%) Qemu Detection: 7,546 (13.38%)
OpenCSG	40 (0.00%)	Big Numbers: 70,798 (53.13%) VMWare Detection: 16,552 (12.42%) VBox Detection: 15,476 (11.61%)
ModelScope	212 (0.31%)	JT 3D Visualization format: 5,046 (75.16%) VBox Detection: 524 (7.80%) Qemu Detection: 524 (7.80%)
OpenMMLab	12 (75.00%)	Is Suspicious: 38 (35.51%) TTA lossless compressed audio: 36 (33.64%) Audio Interchange File Format: 18 (16.82%)
PyTorch Hub	3 (11.53%)	Qemu Detection: 6 (23.08%) VBox Detection: 6 (23.08%) VMWare Detection: 6 (23.08%)

and **VMWare Detection** signatures, which collectively account for more than 95% of all hits. For OpenCSG, the affected repositories are only 40, but they had similar issues as Hugging Face, in addition to the cryptographic malware. For ModelScope's YARA detections are mainly **environmental evasion indicators**, such as **JT 3D Visualization format**, **VBox Detection**, and **Qemu Detection**. Despite the small size of the ecosystem, OpenMMLab exhibits a high smelly repo rate of 75.00%, with common YARA including **Is Suspicious** (35.51%) and audio file signatures such as TTA and AIFF. For PyTorch Hub, 11.53% of repositories contained at least one YARA signature. Its top rules are evenly distributed across **Qemu**, **VBox**, and **VMWare Detection** (23.08% each).

4.3 RQ3: Platform Mitigation Strategies

Table 6 summarizes the security mechanisms of each platform.

4.3.1 *Trust Models and Verification.*

Platforms exhibit three distinct trust paradigms. **Hugging Face**, **ModelScope**, and **PyTorch Hub** follow *trust-all* models where any user can freely upload models. Hugging Face augments this with verified badges for organizational identity (not security audits) [19], while PyTorch Hub shifts trust decisions to users via the `trust_repo` parameter [61]. **OpenMMLab** implements strict *verify-first* with maintainer review of all contributions through pull requests [50]. **OpenCSG** represents a middle ground with *community trust*, allowing open uploads with optional content moderation [44, 47].

Only **Hugging Face** operates comprehensive automated security scanning, triggering on every push with ClamAV (malware), PickleScan (pickle/RCE), TruffleHog (secrets), plus third-party scanners (Protect AI Guardian, JFrog) [16, 17, 22]. However, scans target known patterns rather than comprehensive static analysis, leaving residual RCE risks [18, 27]. **OpenCSG** [48], **ModelScope** [40], and **PyTorch Hub** [61] have no documented platform-level automated scanning. **OpenMMLab** relies on human review without automated scanning [49]. No platform implements comprehensive sandboxing for custom code execution during model loading.

4.3.2 *User-Facing Protections.*

Hugging Face provides the most comprehensive user protections: verified organizational badges [18], prominent UI banners for `trust_remote_code=True`, file badges (ok/infected), a dedicated security documentation hub [21], and SafeTensors support [16]. However, model card code snippets lack inline warnings about `trust_remote_code` risks. **PyTorch Hub** has documentation warnings emphasizing that "models are programs" [57] with interactive prompts via `trust_repo` [59, 61]. **ModelScope** provides only post-download `trust_remote_code` warnings [9, 39]. **OpenCSG** includes community guidelines [46] but lacks explicit `trust_remote_code` warning documentation. **OpenMMLab** has no warnings due to its curated, reviewed model zoo [50, 51].

4.4 RQ4: Developers' Concern

From our systematic analysis of developers' discussions surrounding the `trust_remote_code` mechanism, we derived a taxonomy of concerns observed across forums, issue trackers, and community posts (Figure 5) .

Compatibility and Integration Issues. This category represents discussions where developers report integration failures, incompatibilities, or unexpected behavior when enabling the trust flag. These posts often feature vague complaints such as *"it doesn't work"*, *"the*

Table 6: Comparison of security and trust mechanisms across model-sharing platforms verified against official documentation.

Platform	Upload Verification	Trust Model	Malware Scanning	Warning Systems	User Protection
Hugging Face	Automated scanning [16, 17]	Trust-all with verified badges [19]	Yes – comprehensive multi-layered [6, 18]	UI warnings, file badges (ok/infected), & verified badges [18]	Documentation hub [21], UI warnings, SafeTensors support [16]
OpenCSG	Open uploads via Git or web [47]	Community trust [45]	None documented [46]	None documented [46]	Documentation (community guidelines) [46]
ModelScope	No platform-level automated scanning documented [37]	Trust-all (no verification) [38]	None – no platform-level scanning [37]	Post-download `trust_remote_code` warnings only [9, 39]	Documentation, `trust_remote_code` parameter [39]
OpenMMLab	Maintainer pull-request review [50]	Verify-first [50]	None documented [49]	None documented [49]	Documentation (reviewed code) [51]
PyTorch Hub	No automated verification [58]	Trust-all with `trust_repo` parameter [58]	None documented [58]	Interactive prompts (`trust_repo`), deprecation warnings [58, 59]	Strong documentation warnings: "models are programs" [57]

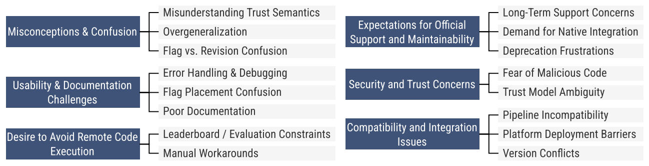

Misconceptions & Confusion
Misunderstanding Trust Semantics
Overgeneralization
Flag vs. Revision Confusion
Usability & Documentation Challenges
Error Handling & Debugging
Flag Placement Confusion
Poor Documentation
Desire to Avoid Remote Code Execution
Leaderboard / Evaluation Constraints
Manual Workarounds
Expectations for Official Support and Maintainability
Long-Term Support Concerns
Demand for Native Integration
Deprecation Frustrations
Security and Trust Concerns
Fear of Malicious Code
Trust Model Ambiguity
Compatibility and Integration Issues
Pipeline Incompatibility
Platform Deployment Barriers
Version Conflicts

Figure 5: Taxonomy of Developers' Concerns.

model fails to load", or *"no support for this flag"*, typically accompanied by minimal debugging information.

- **Pipeline Incompatibility:** A recurring sub-theme involves failures when loading models through pipelines or inference APIs. Because the trust flag is not consistently propagated through these abstractions, developers experience silent failures or partial functionality, requiring non-trivial debugging. These cases often reveal architectural gaps between the core model loaders and downstream pipeline wrappers.
- **Platform Deployment Barriers:** Many developers face difficulties deploying trust-dependent models on managed hosting platforms such as SageMaker, Inference Endpoints, or custom cloud containers. Restrictions on executing remote code, security sandboxing, and the lack of explicit trust flag support in deployment configurations contribute to deployment dead ends.
- **Version Conflicts:** Another dominant sub-pattern involves versioning issues. Breakages often occur due to mismatches between the installed library version and the model's expected environment. Posts frequently cite outdated `transformers` packages, missing backward compatibility, or changes in the trust flag's default behavior across versions. These issues can cascade as small version drifts can break complex pipelines.

Desire to Avoid Remote Code Execution. This category captures discussions where developers explicitly express their reluctance or refusal to enable the trust flag. Unlike compatibility issues, these concerns stem from security or policy perspectives, or from a general mistrust of executing third-party code.

- **Leaderboard / Evaluation Constraints:** In competitive or benchmarking contexts, enabling trust flags is sometimes explicitly forbidden. This restriction stems from fairness, reproducibility, or sandboxing requirements, forcing developers to look for alternative workarounds.
- **Manual Workarounds:** Developers frequently fork repositories, manually download and edit model files, or patch library internals to bypass trust requirements. While these ad-hoc solutions may allow immediate progress, they introduce technical debt, security uncertainty, and maintenance challenges downstream.

Expectations for Official Support and Maintainability. We found discussions that showed expectations from the community for upstream maintainers and platform providers to *"just make it work"*. This category reflects the **expectation gap** between what developers assume the trust flag should offer (automatic, safe, supported execution) and what is actually implemented (manual flag toggling, fragmented support, and inconsistent documentation).

- **Demand for Native Integration:** Developers requested that maintainers integrate model-specific custom code directly into official libraries, thereby removing the need for explicit trust flags. This reflects a preference for official, standardized mechanisms over user-managed trust settings.
- **Deprecation Frustrations:** As the trust mechanism and related APIs evolve, developers face broken pipelines and inconsistent behavior. Complaints in this sub-category often highlight insufficient deprecation notices, breaking changes without migration guides, and a lack of backward compatibility.
- **Long-Term Support Concerns:** Developers working in production environments or regulated domains express concern over whether trust-based model integrations will remain viable in the future. These concerns are often tied to compliance, maintenance, and stability over multiple product cycles.

Misconceptions and Confusion. Not all developer challenges arise from genuine technical limitations. Some stemmed from an incomplete or incorrect understanding of how `trust_remote_code` operates.

- **Flag vs. Revision Confusion:** Developers often conflate the trust flag with revision pinning or version control, mistakenly believing that setting a revision automatically enables trust or vice versa.
- **Misunderstanding Trust Semantics:** Many users incorrectly assume that enabling the flag merely grants permission for metadata loading, not remote code execution. This misinterpretation may lead to underestimating security implications or failing to configure environments correctly.
- **Overgeneralization:** Another common misconception involves assuming that the trust flag behaves uniformly across all model architectures and frameworks. In practice, its support is uneven, leading to mismatched expectations and implementation failures.

Security and Trust Concerns. It represents a distinct and high-stakes theme in developer discourse. Here, developers explicitly reference potential or perceived security risks associated with enabling trust flags. Unlike the "Desire to Avoid RCE" category, which is attitudinal, this category focuses on **explicit threat articulation**.

- **Fear of Malicious Code:** Developers express concerns about arbitrary code execution, supply chain compromises, or untrusted contributors injecting malicious payloads. These discussions frequently reference standard security practices, organization-level security policies, or compliance concerns.
- **Trust Model Ambiguity:** Many developers do not fully understand what "trusting" a model entails at the technical level (e.g., which parts of the repository are executed, what isolation exists, or what verification is done). This lack of transparency fuels suspicion and defensive behavior.

Usability and Documentation Challenges. Even when the trust mechanism works as intended, poor documentation, unclear error messages, or confusing flag placement can create technical barriers.

- **Error Handling & Debugging:** Many developers encounter non-informative or misleading error messages when enabling or failing to enable the trust flag. These debugging hurdles often prolong troubleshooting cycles.
- **Flag Placement Confusion:** Developers frequently struggle to identify where the trust flag should be set (*e.g.*, in CLI arguments, in `pipeline` calls, or at model initialization), especially when documentation is inconsistent across versions.
- **Poor Documentation:** We found posts citing missing, incomplete, or outdated documentation, a lack of minimal working examples, and inconsistent terminology.

5 Discussion

5.1 Ecosystem-wide Security Exposure

Our findings (Table 1) show that the **model-sharing ecosystem is broadly and unevenly exposed to security risks**. While only 2-4% of models on platforms such as Hugging Face, ModelScope, and OpenCSG require custom code, this seemingly small subset represents around *45,000 repositories* containing code executed at load time. Platforms such as OpenMMLab and PyTorch Hub rely entirely on custom code, increasing their systemic attack surface.

Static analysis with Bandit and Semgrep identified two major vulnerability clusters. First, **low-severity but pervasive coding smells** (e.g., CWE-703, B101 `assert` statements) appear across 60–80% of affected repositories, reflecting weak defensive programming practices. Second, **high-impact injection and deserialization vulnerabilities** (e.g., CWE-502, CWE-95, CWE-78) were widespread, particularly on PyTorch Hub and Hugging Face, where dynamic code execution via `pickle` and `eval` is common. Semgrep analysis identified injection and insecure deserialization as the top OWASP categories, underscoring systemic risks of arbitrary code execution at model load time. Notably, the dominance of CWE-117 and CWE-20 in CodeQL results reinforces our observation of *low-severity but pervasive security smells* across the ecosystem. Although CWE-117 issues may seem benign, their combination with insecure cryptographic patterns and unvalidated inputs increases the attack surface. Our CodeQL analysis further underscores the **uneven security exposure across model-sharing platforms** (Table 4). We found a pattern of *cryptographic weaknesses*: nearly all flagged repositories (over 97%) contain "Timing attack against secret" issues, a CodeQL rule that typically signals missing constant-time operations or insecure key handling. While these findings may not always indicate immediately exploitable flaws, their *pervasiveness reflects weak default security hygiene* in model repository codebases.

Importantly, these exposures are not uniform. OpenCSG, despite a low percentage of smelly repos, contributes the highest absolute number of issues due to its massive scale. PyTorch Hub, with a smaller ecosystem, has a disproportionately high rate of high-severity issues, highlighting differences in platform trust boundaries and code review practices.

5.2 Gaps Between Security Mechanisms and Developer Practices

The analysis of platform security mechanisms (Table 6) reveals a **misalignment between available safeguards and how developers interact with model repositories**. Hugging Face, for instance, operates the most advanced malware scanning pipeline, yet *unsafe practices persist widely*, including reliance on `pickle` serialization and unpinned revision loading. The presence of CWE-502 and CWE-95 in hundreds of repositories demonstrates that **technical defenses alone are insufficient to change developer behavior**. Similarly, ModelScope issues warning banners for `trust_remote_code` but lacks sandboxing or pre-upload verification, allowing risky code to propagate. OpenCSG and PyTorch Hub provide minimal automated scanning, relying instead on community trust or basic user prompts (`trust_repo`). The high concentration of injection- and eval-based vulnerabilities in the PyTorch Hub underscores the risks of such lightweight defenses. Furthermore, the *low adoption of secure formats such as Safetensors* (only 6.6% on Hugging Face as of August 2025) shows that safer alternatives are not being embraced at scale, often due to developer inertia, ecosystem lock-in, or lack of clear incentives.

5.3 Results Implications

Our findings have implications for both academia and industry.

For Platform Operators. Platforms must move beyond passive warning systems toward **enforced security boundaries**, including default sandboxing of untrusted custom code, mandatory integrity checks, and stricter upload verification workflows. Richer developer-facing telemetry (e.g., inline vulnerability alerts, dependency provenance) can bridge the gap between automated scanning and practical adoption of secure practices.

For Developers and Maintainers. The results emphasize that developers play a decisive role in the security posture of model hubs. Reliance on `pickle` and eval-based code should be minimized or replaced with safe loading alternatives. Incorporating secure defaults, revision pinning, and code review checklists can help reduce CWE hotspots such as CWE-502 and CWE-95.

For Researchers. Our work showed that though platforms used shared libraries underneath, they have significant differences in handling custom code in model loading. There needs to be work on **automated enforcement frameworks** for trust boundaries, integrating cryptographic integrity verification with runtime isolation.

Our results indicate that tools such as CodeQL can **reveal deep structural weaknesses** in model repository ecosystems that are not surfaced by conventional scanners alone. This creates opportunities for building *automated enforcement frameworks* that couple vulnerability scanning with upload-time checks, runtime sandboxing, and integrity enforcement. Future work can also examine the adoption barriers for secure coding practices, especially around cryptographic operations, to close the gap between warnings and actionable defenses.

5.4 Threats to Validity

Internal Validity. A primary internal threat lies in the accuracy and completeness of our static analysis. Although we employed three well-established tools—Bandit, CodeQL, and Semgrep—to identify security smells and CWE patterns, they may produce false positives or false negatives. However, Siddiq *et al.* show Bandit has 90.79% precision [65]. Semgrep, CodeQL, and YARA are widely used in the research community [2, 14, 31, 33, 41, 42, 66, 67]. Moreover, we manually analyzed discussion posts to conduct open-coding. As mentioned before, this coding was conducted collaboratively by the two authors, whose software development experience ranged from 4 to 12 years, with disagreements resolved by the senior author. The Cohen's kappa score is 0.50, indicating a "moderate" level of agreement [7].

External Validity. Our results may not fully generalize beyond the platforms studied. We focused on five major platforms—Hugging Face, OpenCSG, ModelScope, OpenMMLab, and PyTorch Hub—that dominate the model-sharing ecosystem. For example, Hugging Face hosts around 1.7 million models, and OpenCSG hosts around 200k models. Smaller or private repositories (e.g., enterprise model registries) may exhibit different security characteristics.

6 Related Work

Evolution of Model Hosting Platforms and Pipelines. The evolution from localized model development to centralized sharing platforms constitutes a shift to collaborative ML practices. In the early stages, researchers relied on manual distribution via institutional websites or GitHub repositories, requiring end users to rebuild the entire training and execution environment to reproduce results. The first generation of organized model distribution are mainly *Caffe* (2014) and *TensorFlow Hub* (2018). With *PyTorch Hub* created in 2019, the `torch.hub.load()` interface was also released along with the `trust_repo` parameter [58]. The rapid expansion of *Hugging Face* between 2018 and 2023 further reshaped the landscape: evolving from a simple hosting repository to a fully integrated platform supporting training, inference, and deployment workflows. Yi *et al.* [71] provides a comprehensive analysis of this ecosystem's development, showing how model hubs have become critical infrastructure sustaining millions of models and billions of downloads worldwide. This infrastructural transformation is further quantified by Laufer *et al.* , by analyzing two million models hosted on Hugging Face [34]. Their findings highlight the platform's support for over 4,000 distinct architectures, with an increasing proportion of models depending on custom code to enable advanced functionality beyond standard implementations. Our work focuses on the architectural design of executing code during model loading from the hub.

Quality and security issues of Model Hubs. Jiang *et al.* [30] conducted the first systematic study of these artifacts across eight platforms, revealing that trust relationships in model ecosystems are more implicit and poorly understood than in traditional software. Hu *et al.* . [15] identifies open problems in the LLM supply chain, documenting how fine-tuning workflows, adapter layers, and prompt templates all serve as injection points. Yi *et al.* [71] further characterizes these risks from an edge-computing perspective, showing how LLM-integrated platforms create new trust boundaries among cloud services, edge devices, and end users.

The introduction of `weights_only=True` in PyTorch 1.13 [59] and the `trust_remote_code` flag in Transformers 4.0 [23] represent acknowledgments of the risks, but adoption remains low due to compatibility concerns. Alternative serialization approaches exhibit different trade-offs between security and functionality. SafeTensors, introduced by Hugging Face in 2022 [26], uses a simple header-data format that prevents code execution entirely. The format stores tensors in a flat layout with minimal metadata, enabling zero-copy loading while eliminating executable payloads. However, as our results show and Laufer *et al.* . confirm [34], only 6.6% of models have adopted this format despite platform encouragement. Recent work on secure deserialization provides partial solutions.

While platform owners use scanners to identify vulnerable code and data, Zhao *et al.* 's deployment of *MalHug* [72] identified 91 malicious models and 9 dataset scripts actively exploiting users. JFrog Security Research [29] documented evasion methods that bypass pattern-based scanning, including time-delayed execution, environment fingerprinting, and polymorphic code generation. The August 2025 Protect AI report [55], based on scanning 4.47 million model versions, identifies emerging threats, such as archive slip vulnerabilities and TensorFlow-specific backdoors, that existing tools miss. Our work specifically focuses on the code associated with the model, which is executed during loading, and on developers' concerns about it.

7 Conclusion

Our work provides the first large-scale, cross-platform empirical analysis of remote code execution risks in ML model hosting ecosystems, examining five major platforms. We identified that around 45,000 repositories execute arbitrary code during model loading. Our static analysis evealed most repositories have weak defensive coding practices, and injection and deserialization vulnerabilities (*e.g.*, CWE-502, CWE-95, CWE-78. We also found that most of the malicious code in the category of environmental evasion indicators (*e.g.*, Qemu, VMWare, VBox detections). Although Hugging Face has made significant advances with automated malware scanning pipelines (e.g., ClamAV, PickleScan, Protect AI Guardian), these mechanisms alone are insufficient. Other platforms lack comparable safeguards, with minimal or no sandboxing and weak verification mechanisms. Developer discussions further reveal widespread confusion and misconceptions about trust flags, limited adoption of secure serialization formats like SafeTensors, and tension between usability and security.

References

[1] Anonymous. 2025. Replication Package: Untrusted Model Loading. https://anonymous.4open.science/r/untrusted-model-loading-C8BC/README.md. Accessed: 2025-10-24.

[2] Gareth Bennett, Tracy Hall, Emily Winter, and Steve Counsell. 2024. Semgrep*: Improving the limited performance of static application security testing (sast) tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 614–623.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[4] Beatrice Casey, Kaia Damian, Andrew Cotaj, and Joanna Santos. 2025. An Empirical Study of Safetensors' Usage Trends and Developers' Perceptions. *arXiv preprint arXiv:2501.02170* (2025).

[5] Beatrice Casey, Joanna Santos, and Mehdi Mirakhorli. 2024. A large-scale exploit instrumentation study of AI/ML supply chain attacks in hugging face models. *arXiv preprint arXiv:2410.04490* (2024).

[6] Cisco. 2024. Foundation AI Advances AI Security With Hugging Face. https://blogs.cisco.com/security/ciscos-foundation-ai-advances-ai-supply-chain-security-with-hugging-face. Accessed: 2025-10-10.

[7] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[8] Juliet Corbin and Anselm Strauss. 1990. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques.* Sage Publications.

[9] EvalScope. 2024. FAQ. https://evalscope.readthedocs.io/en/v0.16.3/get_started/faq.html. Accessed: 2025-10-10.

[10] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[11] Mohammad Ghafari, Pascal Gadient, and Oscar Nierstrasz. 2017. Security smells in android. In *2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM).* IEEE, 121–130. doi:10.1109/SCAM.2017.24

[12] GitHub Security Lab. 2025. CodeQL Query Packs. https://securitylab.github.com/tools/codeql/. Accessed: 2025-10-23.

[13] GitHub Security Lab. 2025. CodeQL: Semantic Code Analysis Engine. https://codeql.github.com. Query-based code analysis engine for discovering vulnerabilities. Accessed: October 2025.

[14] Damian Gnieciak and Tomasz Szandala. 2025. Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection. arXiv:2508.04448 [cs.SE] https://arxiv.org/abs/2508.04448

[15] Zuxin Hu, Ning Liu, Tian Zhao, Fan Yang, Xiaowei Deng, Ting Du, Jian Chen, Zongwei Li, and Xiaolong Fan. 2025. Large Language Model Supply Chain: Open Problems from the Security Perspective. In *Proceedings of the 32nd ACM Conference on Computer and Communications Security.* Association for Computing Machinery. doi:10.1145/3713081.3731747

[16] Hugging Face. 2024. 2024 Security Feature Highlights. https://huggingface.co/blog/2024-security-features. Accessed: 2025-10-10.

[17] Hugging Face. 2024. Hugging Face Partners with TruffleHog to Scan for Secrets. https://huggingface.co/blog/trufflesecurity-partnership. Accessed: 2025-10-10.

[18] Hugging Face. 2024. Malware Scanning. https://huggingface.co/docs/hub/en/security-malware. Accessed: 2025-10-10.

[19] Hugging Face. 2024. Organization Verification. https://discuss.huggingface.co/t/organization-verification/17906. Accessed: 2025-10-10.

[20] Hugging Face. 2024. Pickle Scanning. https://huggingface.co/docs/hub/en/security-pickle. Accessed: 2025-10-10.

[21] Hugging Face. 2024. Security. https://huggingface.co/docs/hub/en/security. Accessed: 2025-10-10.

[22] Hugging Face. 2024. Security & Compliance. https://huggingface.co/docs/microsoft-azure/en/security. Accessed: 2025-10-10.

[23] Hugging Face. 2025. Auto Classes. https://huggingface.co/docs/transformers/en/model_doc/auto. Accessed: 2025-10-10.

[24] Hugging Face. 2025. Customizing models — Transformers Documentation. https://huggingface.co/docs/transformers/en/custom_models Accessed: 2025-10-13.

[25] Hugging Face. 2025. Hugging Face Model Hub. https://huggingface.co. Total models: 1,699,968. Accessed: October 2025.

[26] Hugging Face. 2025. Safetensors Documentation. https://huggingface.co/docs/safetensors/index. Accessed: 2025-10-10.

[27] Hugging Face. 2025. Secrets Scanning. https://huggingface.co/docs/hub/en/security-secrets. Accessed: 2025-10-10.

[28] InQuest Labs. 2025. Awesome YARA: A Curated Repository of YARA Rule Sources. https://github.com/InQuest/awesome-yara?tab=readme-ov-file#rules. Accessed: 2025-10-23.

[29] JFrog Security Research. 2024. Examining Malicious Hugging Face ML Models with Silent Backdoors. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/. Accessed: 2025-10-10.

[30] Shangqing Jiang, Nicholas Synovic, Chahat Sethi, Sandeep Indarapu, Parker Hyatt, Marco Schorlemmer, George Thiruvathukal, and James C. Davis. 2022. An Empirical Study of Artifacts and Security Risks in the Pre-trained Model Supply Chain. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses.* Association for Computing Machinery, 105–114. doi:10.1145/3560835.3564547

[31] Bjørnar Haugstad Jåtten, Simon Boye Jørgensen, Rasmus Petersen, and Raúl Pardo. 2025. Scalable Thread-Safety Analysis of Java Classes with CodeQL. arXiv:2509.02022 [cs.SE] https://arxiv.org/abs/2509.02022

[32] Andreas D. Kellas, Neophytos Christou, Wenxin Jiang, Penghui Li, Laurent Simon, Yaniv David, Vasileios P. Kemerlis, James C. Davis, and Junfeng Yang. 2025. PickleBall: Secure Deserialization of Pickle-based Machine Learning Models (Extended Report). https://arxiv.org/abs/2508.15987. arXiv:2508.15987.

[33] Lukas Kree, René Helmke, and Eugen Winter. 2024. Using semgrep oss to find owasp top 10 weaknesses in php applications: A case study. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 64–83.

[34] Benjamin Laufer, Hamidah Oderinwale, and Jon Kleinberg. 2025. Anatomy of a Machine Learning Ecosystem: 2 Million Models on Hugging Face. *arXiv* (2025). arXiv:2508.06811 [cs.SI] https://arxiv.org/abs/2508.06811 Accessed: 2025-10-10.

[35] Victor Alvarez Martín. 2014. YARA: The pattern matching swiss knife for malware researchers. https://virustotal.github.io/yara/. Accessed: 2025-10-23.

[36] MITRE Corporation. 2025. Common Weakness Enumeration (CWE). https://cwe.mitre.org. A community-developed list of common software and hardware weakness types. Accessed: October 2025.

[37] ModelScope. 2024. ModelScope: Bring the Notion of Model-as-a-Service to Life. https://github.com/modelscope/modelscope. Accessed: 2025-10-10.

[38] ModelScope. 2024. Phi-3-mini-128k-instruct. https://modelscope.cn/models/LLM-Research/Phi-3-mini-128k-instruct. Accessed: 2025-10-10.

[39] ModelScope. 2024. Releases. https://github.com/modelscope/modelscope/releases. Accessed: 2025-10-10.

[40] ModelScope. 2025. ModelScope Model Hub. https://modelscope.cn. Total models: 68,736. Accessed: October 2025.

[41] Nitin Naik, Paul Jenkins, Roger Cooke, Jonathan Gillett, and Yaochu Jin. 2020. Evaluating automatically generated YARA rules and enhancing their effectiveness. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI).* IEEE, 1146–1153.

[42] Nitin Naik, Paul Jenkins, Nick Savage, Longzhi Yang, Tossapon Boongoen, Natthakan Iam-On, Kshirasagar Naik, and Jingping Song. 2021. Embedded YARA rules: strengthening YARA rules utilising fuzzy hashing and fuzzy rules for malware analysis. *Complex & Intelligent Systems* 7, 2 (2021), 687–702.

[43] OpenAI. 2023. GPT-4 Technical Report. arXiv preprint arXiv:2303.08774.

[44] OpenCSG. 2024. csghub-server. https://github.com/OpenCSGs/csghub-server. Accessed: 2025-10-10.

[45] OpenCSG. 2024. Information about the OpenCSG community. https://github.com/OpenCSGs/community. Accessed: 2025-10-10.

[46] OpenCSG. 2024. OpenCSG Documentation Center. https://www.opencsg.com/docs/en/. Accessed: 2025-10-10.

[47] OpenCSG. 2024. Uploading Codes. https://www.opencsg.com/docs/en/code/upload_codes. Accessed: 2025-10-10.

[48] OpenCSG. 2025. OpenCSG Model Hub. https://opencsg.com. Total models: 192,556. Accessed: October 2025.

[49] OpenMMLab. 2024. Benchmark and Model Zoo - MMDetection's documentation. https://mmdetection.readthedocs.io/en/latest/model_zoo.html. Accessed: 2025-10-10.

[50] OpenMMLab. 2024. Contribution Guide – MMDetection3D 1.4.0 documentation. https://mmdetection3d.readthedocs.io/en/latest/notes/contribution_guides.html. Accessed: 2025-10-10.

[51] OpenMMLab. 2024. OpenMMLab Detection Toolbox and Benchmark. https://github.com/open-mmlab/mmdetection. Accessed: 2025-10-10.

[52] OpenMMLab. 2025. OpenMMLab Model Zoo. https://openmmlab.com. Total models: 16. Accessed: October 2025.

[53] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155* (2022).

[54] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. 2022. Code Smells Detection and Visualization: A Systematic Literature Review. *Archives of Computational Methods in Engineering* 29, 1 (Jan. 2022), 47–94. doi:10.1007/s11831-021-09566-x

[55] Protect AI. 2025. 4M Models Scanned: Hugging Face + Protect AI Partnership Update. https://protectai.com/blog/hugging-face-protect-ai-six-months-in. Accessed: 2025-10-10.
[56] PyCQA. 2025. Bandit: Security Linter for Python Source Code. https://bandit.readthedocs.io. Static analysis tool for detecting common security issues in Python code. Accessed: October 2025.
[57] PyTorch. 2024. Security Policy - pytorch/pytorch. https://github.com/pytorch/pytorch/security. Accessed: 2025-10-10.
[58] PyTorch. 2024. torch.hub – PyTorch 2.8 documentation. https://pytorch.org/docs/stable/hub.html. Accessed: 2025-10-10.
[59] PyTorch. 2024. torch.load(..., weights_only=True) currently raises a warning. https://github.com/pytorch/pytorch/issues/52181. Accessed: 2025-10-10.
[60] PyTorch. 2025. torch.hub — PyTorch Documentation. https://docs.pytorch.org/docs/stable/hub.html Accessed: 2025-10-13.
[61] PyTorch Contributors. 2025. pytorch/pytorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://github.com/pytorch/pytorch. Accessed: 2025-10-21.
[62] r2c. 2025. Semgrep: Lightweight Static Analysis for Modern Languages. https://semgrep.dev. Open-source static analysis tool supporting pattern-based security and compliance checks. Accessed: October 2025.
[63] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 164–175. doi:10.1109/ICSE.2019.00033
[64] Md Rayhanur Rahman, Akond Rahman, and Laurie Williams. 2019. Share, But be Aware: Security Smells in Python Gists. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 536–540. doi:10.1109/ICSME.2019.00087
[65] Mohammed Latif Siddiq, Shafayat Hossain Majumder, Maisha Rahman Mim, Sourov Jajodia, and Joanna C.S. Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*.
[66] Mohammed Latif Siddiq and Joanna C. S. Santos. 2022. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security* (Singapore, Singapore). Association for Computing Machinery, New York, NY, USA, 29–33. doi:10.1145/3549035.3561184
[67] Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy, and Anna Muller. [n. d.]. SALLM: Security Assessment of Generated Code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24)* (Sacramento, CA, USA) *(ASEW '24)*. 12 pages. doi:10.1145/3691621.3694934
[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.
[69] VirusTotal. 2025. YARA: Pattern Matching Swiss Knife for Malware Researchers. https://github.com/virustotal/yara?tab=readme-ov-file#additional-resources. Accessed: 2025-10-23.
[70] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6
[71] Wenxin Yi, Dinh Phung Luu, Kangyu Ma, Jiayi Wang, Xiaowei Deng, Ruoxi Fu, Jiajing Chen, Zongwei Li, Wenxuan Yu, Yue Wang, Philip Khang, Fan Yang, Yue Zhang, and Yuanyuan Fang. 2024. Characterizing and Understanding the Risks of Large Language Model-Integrated Platforms: A Supply-Chain Perspective on the Security of Edge LLM Systems. *arXiv* (2024). arXiv:2409.09368 [cs.CR] https://arxiv.org/abs/2409.09368 arXiv:2409.09368v1.
[72] Jian Zhao, Shenao Wang, Yanjie Zhao, Xinyi Hou, Kailong Wang, Peiming Gao, Yuanchao Zhang, Chen Wei, and Haoyu Wang. 2024. Models Are Codes: Towards Measuring Malicious Code Poisoning Attacks on Pre-trained Model Hubs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2087–2098. doi:10.1145/3691620.3695271