Execution-as-Configuration: Security Smells in Model Configuration Artifacts

Mohammed Latif Siddiq
msiddiq3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Prince Noah Johnson
pjohns24@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Abstract

Model sharing hubs rely on configuration artifacts to control model loading and execution. Although treated as metadata, files such as config.json determine class resolution and dependency selection, effectively making configurations an execution control surface. Yet their security implications remain largely unexplored. In this paper, we present a large-scale study of model configuration-related security smells. We introduce a taxonomy of 12 security smell types capturing execution-enabling constructs, trust boundary confusion, supply-chain risks, information leakage, and operational risks in model configuration files. To operationalize this taxonomy, we design MONTICELLO, a repository-aware security smell detection approach that combines pattern-based rules with contextual reasoning. We analyze 44,812 repositories across Hugging Face, OpenCSG, and ModelScope. Configuration smells are pervasive: 65.5%–95.5% of repositories contain at least one. These smells expose repositories to concrete risks, including unsafe deserialization, dynamic code execution, and trust boundary violations.

CCS Concepts

• **Software and its engineering** → **Software verification and validation**; *Software system structures*; • **Security and privacy** → *Software and application security*.

Keywords

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

ACM Reference Format:

Mohammed Latif Siddiq, Prince Noah Johnson, and Joanna C. S. Santos. 2026. Execution-as-Configuration: Security Smells in Model Configuration Artifacts. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837470

1 Introduction

Large Language Models (LLMs) are increasingly embedded in applications and developer workflows, supporting tasks such as code generation and data analysis [5, 34, 36]. Modern LLMs are rarely used in isolation, but instead reused, fine-tuned, and composed into software pipelines that integrate model artifacts, configuration files, and repository-defined code [48]. *Model hubs* (*e.g.*, Hugging Face [18]) power this ecosystem by hosting pre-trained models and datasets at scale [55] (for example, Hugging Face has over 2.1 million models as of Jan. 2026). While model hubs democratize access to state-of-the-art ML and speed up deployment [36, 55], they also introduce *AI/ML supply-chain risks* [43].

This shift is driven by the increasing reliance on configuration files to control model execution. Frameworks such as *transformers* use configuration files to determine class instantiation, code paths, and imported repository modules. These artifacts govern execution decisions, effectively making configuration a first-class *execution control surface*. We introduce the notion of *Execution-as-Configuration*, where configuration artifacts implicitly control code loading and execution behavior, representing a shift in the security model of model hubs: execution logic is partially encoded in configuration rather than source code. Our formulation is conceptually related to prior notions such as *Configuration-as-Code* [42] and *Infrastructure-as-Code* [41], but differs in that configuration artifacts directly influence runtime execution and dynamic code loading. A key mechanism underlying this paradigm is the presence of *configuration-to-code bridges* (*e.g.*, *auto_map*) which resolve configuration fields into executable Python classes, enabling indirect code execution, cross-repository dependencies, and stealth expansion of the trusted computing base. Such execution may occur even when explicit safeguards (*e.g.*, *trust_remote_code=False*) are present, due to implicit resolution logic in the loading framework.

Prior work [6, 7, 21, 43, 48, 57] has primarily focused on risks from *unsafe deserialization* (*e.g.*, Python `pickle`) and the execution of untrusted code during model loading, motivating safer formats such as `safetensors` [6]. Recent empirical studies further highlight widespread reliance on custom model loading and unsafe defaults (*e.g.*, `trust_remote_code`), exposing systems to remote code execution risks [48]. As these serialization and loading risks are increasingly recognized and mitigated, the attack surface shifts to the *configuration layer*, where seemingly benign configuration entries can trigger dynamic code loading, import arbitrary modules, or introduce implicit dependencies on external repositories.

Despite these risks, configuration-driven execution remains largely unexplored. Existing security analyzers [44, 51, 53] treat JSON and YAML files as passive descriptors, overlooking their execution semantics. This is concerning because configuration files are processed early in the model-loading pipeline and may bypass traditional security controls, enabling AI/ML supply-chain attacks. Therefore, we present the first large-scale empirical study of *configuration and repository-aware security smells* in modern model hubs. We introduce a novel taxonomy of 12 configuration-smell types

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837470

that capture execution-enabling constructs, trust-boundary confusion, supply-chain expansion, information leakage, and operational instability. Moreover, we design MONTICELLO, an approach that combines reusable predicates with contextual reasoning to detect execution-relevant smells in model configurations. As such, this paper makes the following contributions:

- We conduct a systematic analysis of configuration artifacts in model-sharing platforms and present **a taxonomy of 12 configuration security smells** (§ 5.2).
- **An approach (MONTICELLO) that detects execution-relevant configuration smells** with high precision by analyzing configuration fields and their usage across repositories (§ 6.1).
- **A large-scale empirical study of 44,812 repositories with configuration artifacts across Hugging Face, ModelScope, and OpenCSG**. Our results show that configuration smells are pervasive, with 65.5%–95.5% of repositories containing at least one smell, and that a small number of constructs (notably `auto_map`) dominate the attack surface (§ 6.5).
- **An analysis of 31,455 Python files transitively reachable through configuration-to-code bridges using three static analyzers**. We show that configuration smells often expose repositories to concrete vulnerabilities, *e.g.*, unsafe deserialization, and dynamic code execution (§ 7.2).

2 Background & Related Work

2.1 Custom Model Loading in Model Hubs

Model-sharing platforms host pre-trained models, datasets, and associated tooling, enabling developers to rapidly reuse, fine-tune, and deploy state-of-the-art models across diverse applications [52, 55]. Early approaches relied on manually sharing model code and weights, requiring users to reconstruct architectures and execution environments from documentation [52]. Rather than distributing only static weight files, contemporary model repositories typically include auxiliary artifacts that define how models are instantiated, configured, and integrated into downstream pipelines.

To support architectural diversity, these platforms allow ***custom model loading***, enabling developers to provide Python code executed during model initialization to implement custom architectures, tokenizers, pipelines, or configuration logic. Popular frameworks (*e.g.*, *transformers*) dynamically resolve model classes, import repository-defined modules, and construct execution pipelines at runtime [36, 52]. This enables rapid model innovation but expands the trusted computing base beyond core framework code.

The central mechanism is the `auto_map` field in `config.json`. When a consumer calls a loader such as `AutoModel.from_pretrained`, the framework fetches and parses the repository's `config.json`. If an `auto_map` entry is present (*e.g.*, `"auto_map":"AutoModelForCausalLM": "modeling_custom.MyModel"`), `transformers` downloads the referenced Python file (*e.g.*, `modeling_custom.py`), caches it to disk, imports the module, and instantiates the named class (`MyModel`). This entire sequence, network fetch, disk write, dynamic import, and instantiation, is triggered by a single JSON field and executes *before* the model weights are loaded. When the value contains a cross-repository prefix (*e.g.*, `"AutoModelForCausalLM":"other-org/other-repo--module.Class"`), the same call chain fetches code from *that* repository instead, silently extending the trust boundary beyond the model being loaded. This makes `auto_map` an execution control surface rather than passive metadata.

As shown in Listing 1, the `architectures` field declares the model class, `auto_map` redirects framework loaders to repository-defined Python modules, and `custom_pipelines` registers a custom inference pipeline that is dynamically imported at load time, all three trigger code loading during `from_pretrained`, which could be misused by an attacker to execute malicious code.

Listing 1 Execution-relevant fields in a model `config.json`.

config.json

```
{"architectures": ["MyCustomModel"],
"auto_map": {
  "AutoConfig":"org/model--config.MyConfig", "AutoModelForCausalLM":"org/model--model.MyModel"
},
"custom_pipelines": {
  "text-generation": { "impl": "pipeline.MyPipeline", "pt": "AutoModelForCausalLM"}
}}
```

2.2 Security Risks in Model Hubs

Expanding the trusted computing base beyond core frameworks introduces significant security risks in model-sharing ecosystems. Prior work shows that trust relationships are often implicit, leading to widespread reuse of unverified artifacts [19]. This has enabled real-world attacks, including unsafe deserialization, malicious model injection, and remote code execution [6, 7, 48, 56].

Frameworks provide partial mitigations, such as the `trust_remote_code` flag and safer formats like *safetensors*. However, these safeguards primarily govern explicit code execution. Configuration artifacts are processed *earlier*, determining class resolution and dependencies, often before such controls are enforced [55]. This places configuration handling at a critical trust boundary, resembling the *confused deputy problem* [15]. Recent work spans both detection and exploitation. Zhao *et al.* [56] propose *MalHug*, a static analysis pipeline for detecting malicious models. Complementing this, our work employs static analysis to identify smelly configurations that can control execution behavior during model loading and to assess the effectiveness of existing defenses.

2.3 Configuration-Driven Code Execution

Model configuration files (*e.g.*, *config.json*) encode essential metadata used by model loaders. These artifacts specify architectural parameters, tokenizer behavior, generation defaults, dependency paths, and mappings from symbolic identifiers to concrete Python classes [32, 52, 55]. As model hubs evolved, configuration artifacts have become more complex, encoding references to model classes, external modules, and pipeline behaviors that determine which code is dynamically loaded and executed at runtime [52, 54]

While historically treated as passive descriptors, configuration artifacts increasingly participate directly in execution. Fields such as `auto_map`, `architectures`, and `custom_pipelines` determine which classes are dynamically imported and which repository-defined modules are executed during model loading [7]. Adapter and composition mechanisms further introduce indirection by allowing configurations to reference external base models or remote repositories [6]. As a result, configuration files form a *configuration-to-code bridge*: metadata entries are interpreted by trusted loaders to resolve import paths, fetch remote dependencies, and invoke repository-defined code. This design blurs the boundary between

data and code and elevates configuration artifacts to a first-class control surface for execution [7, 55].

From a security perspective, this architecture introduces layered trust boundaries among framework code, platform infrastructure, model maintainers, and consumers. Recurring config-level anti-patterns, such as overly permissive class mappings and unpinned remote dependencies, expand the trusted computing base and weaken isolation guarantees [33, 58]. ConfigScan [11] uses LLM-based analysis to detect malicious configurations in AI model repositories; however, it focuses on concrete attacks and does not systematically characterize the broader class of configuration anti-patterns that can silently expand the trusted computing base through mechanisms such as `auto_map`, cross-repository class mappings, and unpinned revisions. We refer to this paradigm as ***Execution-as-Configuration***, and we define ***configuration security smells*** as recurring config-level patterns in model-loading artifacts that indicate security weaknesses and can lead to security breaches [41, 47]. While a configuration security smell does not always lead to a security breach, it deserves inspection. These smells are independent of whether the referenced code is itself malicious; rather, they signal risky patterns in how models are loaded.

3 Threat Model

We study an underexplored security problem in model-sharing ecosystems: *configuration artifacts controlling execution.*

Stakeholders. Model loading introduces trust relationships among three stakeholders: *model creators*, *platform maintainers* (*e.g.*, Hugging Face, ModelScope, OpenCSG), and *model consumers*. Our threat model captures attacks that exploit unsafe defaults, inconsistencies, or misconfigurations to trigger unintended execution, expand trust boundaries, or introduce supply-chain risks.

Adversary Assumptions. Similar to prior works [7, 48], we assume adversaries can upload or modify repositories on model platforms, either as malicious contributors or through compromised maintainer accounts (*e.g.*, via leaked `HF_TOKEN`). Attackers may also leverage community features to disseminate malicious artifacts. Moreover, we consider platform-mediated risks, such as insecure defaults (*e.g.*, `trust_remote_code=True`) and automated mirroring of unvetted models into trusted environments.

Threat Scenarios. We consider four representative threat scenarios arising from configuration-driven execution. **(1)** *Malicious configuration injection* in which attackers manipulate execution-relevant fields to redirect class resolution to attacker-controlled modules, enabling arbitrary code execution without modifying source files. Concretely, an attacker who controls a repository `evil-org/backdoor` can craft a `config.json` containing `"auto_map":{"AutoModelForCausalLM":"evil-org/backdoor--payload.Exploit"}`, where `payload.py` defines a class whose `__init__` exfiltrates environment variables or installs a reverse shell. When a consumer calls `AutoModel.from_pretrained` on any model whose configuration references this entry, the framework codepayload.py from the attacker's repository, imports it, and instantiates `Exploit`, executing the malicious code before model weights are even loaded, and potentially before `trust_remote_code` is checked. **(2)** *Configuration poisoning* in which compromised repositories introduce subtle configuration changes that trigger malicious behavior while appearing trustworthy. **(3)** *Transitive dependency exploitation*, which occurs when configuration fields (*e.g.*, `base_model_name_or_path`) introduce implicit dependencies on attacker-controlled or unpinned models, enabling supply-chain compromise. **(4)** *configuration drift* (time-of-check-to-time-of-use; TOCTOU), which arises when configurations reference external repositories that later become malicious, leading to deferred-execution vulnerabilities. Beyond deliberate attacks, unintentional configuration smells (*e.g.*, unbounded generation limits or hard-coded secrets) can also be exploited for denial-of-service or credential leakage.

Trust Relationships. Model consumers implicitly trust configuration files to safely describe execution behavior. However, configuration metadata determines *which code is executed*, which can bypass user intent and platform safeguards.

Motivating Example. These threats are not hypothetical. CVE-2026-45829 [1] demonstrates how configuration smells propagate through the supply chain: an unauthenticated attacker sends a ChromaDB server a request whose embedding-function configuration points `model_name` to an attacker-controlled Hugging Face repository whose `config.json` carries `auto_map` and `trust_remote_code: true`; the server loads the model and executes the payload before its own authentication check runs. Similarly, CVE-2026-5241 [2] showed that a model creator can embed `trust_remote_code: true` directly in `config.json`, silently overriding the caller's explicit `trust_remote_code=False` setting during nested model loading. These incidents showcase the motivation of our work: ***configuration files act as implicit execution vectors that can bypass both user intent and platform-level safeguards***. Subtle configuration security smells can transform benign model artifacts into delivery mechanisms for AI/ML supply-chain attacks.

4 Research Questions

RQ1 What types of security risks arise from configuration-driven execution in model-sharing platforms?

In this RQ, we conduct a qualitative analysis of public model configuration files to derive *a novel taxonomy of configuration security smells*. This taxonomy captures execution-enabling constructs, trust boundary confusion, supply chain expansion, information leakage, and operational instability, providing a structured vocabulary for reasoning about configuration-driven security risks.

RQ2 How prevalent are configuration security smells across model hub repositories?

We develop a detector of security smells in configuration files and use it to scan public repositories across various platforms. Our goal is to quantify the prevalence, distribution, and co-occurrence patterns of configuration security smells. This RQ reveals how often configuration files act as execution control surfaces and expand the attack surface of ML-enabled systems.

RQ2 What security issues exist in Python code reachable through configuration-to-code bridges?

Configuration artifacts frequently reference repository-defined Python modules via entries such as *auto_map*. In RQ2, we resolve

these configuration-to-code bridges and analyze the transitively referenced Python files using static security analyzers. We study the extent to which configuration-driven execution exposes repositories to traditional code-level vulnerabilities and security smells.
Figure 1 shows the overview of the study, while the next section describes the methods and results for each RQ.

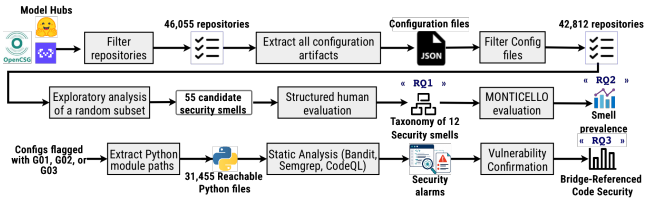

Figure 1: Overview of the study.

5 RQ1: Security Smells in Model Config.

5.1 Methodology

5.1.1 Dataset Collection. We build our dataset on top of the corpus collected in a prior large-scale study of remote code execution in model hosting ecosystems [48]. That dataset provides a comprehensive snapshot of recent model repositories that allow user-managed loading logic and configuration-driven execution, which forms the foundation for our configuration security smell analysis.

Model Sharing Platform Selection. This prior dataset [48] was created by examining a recent list of 15 popular model-sharing platforms compiled by Zhao *et al.* [55] and applying a set of inclusion criteria to determine platform suitability. We adopt their criteria (1)-(3) [55] and introduce two new criteria, (4) and (5), to account for configuration-driven execution. Specifically, we include a platform in our work only if it satisfies *all* of the following: (1) it is publicly accessible; (2) it provides practical means to retrieve metadata and files from all hosted model repositories (*e.g.*, via APIs or web crawling); (3) models can be programmatically fetched and instantiated through standard loading APIs provided by `transformers`, `torch.hub`, or their wrappers, without requiring reimplementation; (4) it permits the execution of user-managed code during model loading; and (5) model loading behavior is primarily defined by user-managed repositories rather than centralized frameworks, exposing a key *trust boundary* that enables configuration-driven code execution and thus introduces greater security risk, the focus of this work. These criteria ensure that configuration files play an active role in controlling execution and that loading semantics are not fully centralized by the hosting framework. After applying these criteria, we identify **three** platforms that satisfy these conditions: **Hugging Face** [18], **OpenCSG** [35], and **ModelScope** [31].

Model Repository Selection. We collect metadata (*e.g.*, URLs, tags, and list of files) for all hosted model repos in the selected platforms. We then identify repositories whose loading behavior is influenced by configuration artifacts and user-managed code. A repository is included if it satisfies *any* of the following: **(i)** tagged with `custom_code`; **(ii)** contained entry-point files (*e.g.*, `tokenizer.py`, `__init__.py`, `hubconf.py`); or **(iii)** included *.py* files prefixed with `modeling_`, `tokenization_`, or `configuration_`. These criteria follow the loading mechanisms defined in *transformers* [17] and *torch.hub* [40]. This resulted in 36,697 repositories from Hugging Face, 6,165 from OpenCSG, and 3,193 from ModelScope.

Configuration Files Extraction. For each repository, we extract all configuration artifacts used for model loading and execution (*e.g.*, `config.json`, `adapter_config.json`, `tokenizer_config.json`, *etc.*), as well as pipeline- and framework-specific JSON configuration files. We retain only repositories that contain at least one syntactically valid, non-empty, parseable load-time configuration file, and discard repositories with malformed or non-JSON configurations. This ensures that all subsequent analysis is performed on well-formed config files that directly influence model loading execution.

5.1.2 Open Coding for Smell Discovery.

Initial Smell Identification. To identify security smells and derive a data-driven taxonomy, we adopted an open coding methodology inspired by grounded theory [9]. One author (with over 6 years of experience in ML and software security) conducted an exploratory analysis of a random subset of **2,240** configuration files (approximately 5% of the 44,812 repositories, consistent with prior studies [41]). During this process, the selected configuration files were iteratively inspected to identify recurring, security-relevant patterns, particularly those affecting code execution, trust boundaries, and external dependencies. Identified patterns were annotated and grouped based on similarity and shared risk characteristics. After iterative refinement and deduplication, this process led to an initial set of **55 candidate security smells**.

Validation and Refinement. To validate and refine these candidates, we conducted a structured human evaluation with a second annotator (3 years of ML/Software Security experience). Prior to independent annotation, the two annotators completed a calibration round on a held-out set of 10 configuration files (disjoint from the validation set) using a shared codebook. The codebook included (i) formal definitions of each candidate smell, (ii) three positive and two negative hand-crafted examples per smell, and (iii) decision rules for ambiguous cases (*e.g.*, distinguishing tokenizer-vocabulary fields from credential-indicative field names, or determining whether a high `max_new_tokens` value reflects a legitimate long-context model versus an unbounded default). During calibration, annotators discussed disagreements and refined the decision rules until both reached consistent judgments on the held-out set; the codebook was updated accordingly before the main annotation began. The two annotators then independently annotated a validation set of **109** configuration instances (two representative samples per smell)[1]. For each *configuration-smell* pair, annotators labeled whether the smell was PRESENT (TRUE) or ABSENT (FALSE) according to the calibrated codebook. Disagreements were resolved through adjudication by a third author (10+ years of Security / ML experience), resulting in the final ground-truth labels.
We assessed inter-rater agreement using Cohen's κ [8] and Krippendorff's α [23]. Across the 109 config-smell pairs (P(TRUE) = 73.4%), we obtained $\kappa = 0.316$ and $\alpha = 0.248$ (*fair agreement*). The moderate agreement reflects the inherent ambiguity of early-stage

[1] One smell category is represented by a single sample due to limited availability.

taxonomy construction, where smell boundaries are not yet stabilized and annotators must judge novel, domain-specific patterns without established precedent. Such agreement levels are consistent with prior work on taxonomy development in software engineering [41]. More importantly, the calibration round and subsequent adjudication by the senior author ensured that all final labels reflect consensus, and the iterative consolidation from 55 to 12 smell types further resolved the boundary ambiguities that drove initial disagreement. These types were defined based on shared semantic properties, and were subsequently mapped to relevant Common Weakness Enumeration (CWE) entries by the senior author, who has prior experience in CWE assignment.

Figure 2 illustrates our qualitative coding process for deriving smell groups. Starting from raw configuration snippets, we first extract relevant key-value pairs and assign initial labels through open coding, capturing fine-grained security-relevant patterns (*e.g.*, secret-like values and credential-indicative field names). We then perform axial coding to group semantically related patterns based on their shared security implications, such as the exposure of authentication material. In this example, two initial patterns are merged into the category **G09-Credential Exposure**, as both may lead to the leakage of sensitive credentials despite syntactic differences.

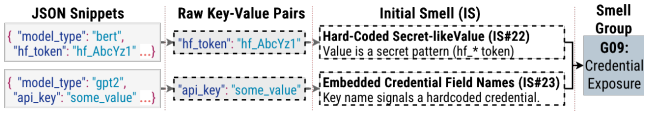

Figure 2: Qualitative security smell identification example.

5.2 A Taxonomy of Model Config. Smells

Our analyses led to the creation ***a taxonomy of security smells in model loading configuration files***. As shown in Figure 3, this taxonomy is organized around common weaknesses (CWEs). We describe each security smell below, and an example of each is provided in Figure 3. Unless otherwise noted, these smells are not limited to `config.json` files; they may also arise in other execution-relevant configuration artifacts such as `adapter_config.json`, `tokenizer_config.json`, and `generation_config.json`.

5.2.1 Input-Based Class/Code Selection (CWE-470). Two security smell types are related to configuration parameters indirectly controlling class selection or execution logic. This allow attackers to manipulate control flow to trigger malicious behavior [24].

G01 – Auto Class Remapping. It occurs when a configuration file uses the `auto_map` field to redirect a standard `Auto*` class to a repository-defined Python implementation. Although such remapping is a flexible extensibility mechanism in the `transformers` ecosystem, it introduces risk by executing repository-provided Python code during loading, rather than only deserializing weights or metadata. Consequently, the configuration does not merely describe a model; it also acts as an instruction to import and instantiate code whose behavior may include arbitrary filesystem access, dependency loading, or network interaction.

Impact and conditions. It enables credential exfiltration, backdoor installation, or silent data poisoning at model load time, without modifying the model weights. The severity depends on whether the consumer's environment enforces `trust_remote_code=False` and on whether downstream applications validate `auto_map` targets before loading (as CVE-2026-45829 [1] demonstrated, a malicious `auto_map` entry combined with `trust_remote_code: true` in *config.json* enabled pre-authentication RCE in a downstream application). As shown in Table 5, the code loaded via `auto_map` often contains unsafe deserialization (CWE-502) and eval injection (CWE-95).

G02 – Custom Pipeline Entry Point. When a configuration registers a non-default inference pipeline through the `custom_pipelines` field, the pipeline class is dynamically imported and constructed during loading. This expands the trusted code base from the framework itself to repository-supplied Python code. Thus, a configuration that appears to specify only an inference task also controls which code is executed to implement that task.

Impact and conditions. An attacker who controls the referenced module can execute arbitrary code whenever a consumer invokes the pipeline for inference, including in production environments. The risk is amplified when the pipeline name matches a common task (*e.g.*, *text-classification*), since consumers may not suspect that a familiar task name resolves to custom code rather than a framework built-in. Static analysis of code reachable through `custom_pipelines` reveals the same pattern: pickle usage and dynamic evaluation dominate findings across all three platforms (Table 5).

5.2.2 Inclusion of Functionality from Untrusted Sources (CWE-829). Four security smell types occur when configurations reference or incorporate executable functionality (*e.g.*, libraries, adapters, or remote code) from sources outside the system's trusted control sphere [30]. This introduces a security risk when the provenance and integrity of external components are not strictly enforced because configurations may implicitly trust remote or cross-repository resources, thereby increasing the likelihood of incorporating unverified or malicious functionality.

G03 – Cross-Repository AutoMap. It arises when an `auto_map` entry refers to code hosted in a different repository, typically using a value of the form `other-org/other-repo–module.Class`. This is a security smell because it silently extends the trust boundary beyond the repository being inspected: loading one model may execute code maintained by a different party in a separate repository.

Impact and conditions. This smell introduces a supply-chain dependency that is not obvious from the repository contents alone. If the external repository is compromised, deleted, or taken over (*e.g.*, via namespace squatting [33], in which an attacker registers a repository under an abandoned or similarly named organization), every model that references it becomes a vector for supply-chain code execution. The risk increases when the referenced repository belongs to a different organization and no `revision` is pinned (§5.2.3), allowing the resolved code to change without modifying the referencing configuration.

G04 – Parameter-Efficient Fine-Tuning (PEFT) Adapter Configuration. It occurs when an adapter configuration declares a PEFT mechanism (*e.g.*, LoRA) through fields like `peft_type` together with a referenced base model. This is a security smell because the adapter itself is not self-contained: using it requires resolving and loading the referenced base model, which may in turn trigger its own

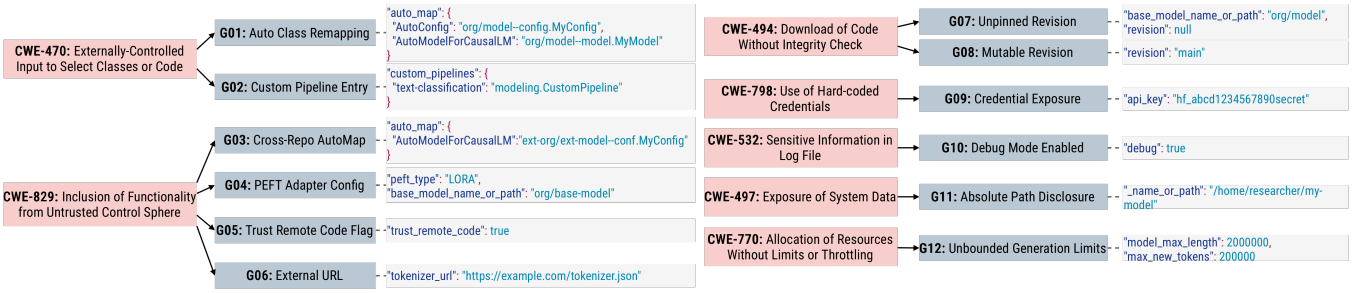

Figure 3: Taxonomy of Model Loading Config Smells

execution-relevant configuration and code-loading behavior. That is, the adapter configuration creates an indirect execution path and an implicit cross-repository dependency, even if it does not itself contain fields such as `auto_map` or `trust_remote_code`.

Impact and conditions. It can transitively trigger the execution surface of its base model, including any smells (*e.g.*, G01, G05) that the base model carries. The impact depends on the configuration of the base model repository: if the base model uses `auto_map` or has `trust_remote_code` set, those risks are silently inherited by the adapter's consumers.

G05 – Trust Remote Code Flag. When a configuration sets `trust_remote_code=true`, it disables a key safety guard, allowing the execution of repository-provided Python code that would otherwise be blocked by the framework. While the flag may be used intentionally for legitimate custom models, its presence signals that loading the model depends on trusting unreviewed remote code, and that downstream users may unknowingly inherit this trust decision.

Impact and conditions. It allows arbitrary Python code execution during loading, making the repository functionally equivalent to an untrusted script download. The risk is compounded when the flag is set by the *model creator* in a configuration file rather than by the consumer at invocation time, since downstream users who load the model may not be aware they have opted into remote code execution. The practical severity depends on whether the consumer's loading pipeline re-checks or overrides the flag before executing repository code; as CVE-2026-5241 [2] demonstrated, a model creator can embed `trust_remote_code` in `config.json` such that it silently overrides the caller's explicit `trust_remote_code=False` setting. When this flag is active, the loaded code is subject to the vulnerabilities cataloged in Table 5.

G06 – External URL in Configuration. When configurations embed external URLs fetched at initialization/load time, it expands the set of external resources that influence execution beyond the repository itself. Depending on how the referenced content is handled, the system may download untrusted artifacts, introduce additional supply-chain dependencies, or trigger unsafe behaviors.

Impact and conditions. Embedded URLs can be exploited for server-side request forgery (SSRF), data exfiltration via URL-based callbacks, or injection of malicious artifacts at initialization time. The severity depends on the fetching context: URLs fetched inside a privileged environment (*e.g.*, a cloud-hosted inference endpoint with access to internal services) carry higher risk than those fetched on a developer's local machine. URLs pointing to attacker-controlled domains are directly exploitable, while those pointing to legitimate but unpinned resources share the time-of-check-to-time-of-use (TOCTOU) risks described in G07–G08 (§5.2.3 - 5.2.3).

5.2.3 Download of Code Without Integrity Check (CWE-494). Two security smell types (G07,G08) are due to referencing external models or code without immutability and integrity guarantees [25].

G07 – Unpinned Revision. When a configuration references an external model or dependency without pinning a fixed revision (*i.e.*, `revision` is absent or `null`), it allows the resolved artifact to change over time. This introduces time-of-check-to-time-of-use (TOCTOU) risks and undermines reproducibility and auditability.

Impact and conditions. An attacker who gains write access to the referenced repository can silently replace the resolved artifact, compromising all downstream consumers without any visible configuration change. The risk is highest when combined with G03 or G04, since the unpinned dependency is in a repository outside the consumer's control.

G08 – Mutable Revision Reference. When a configuration sets `revision` to a mutable reference such as `main`, `HEAD`, or a pull-request ref., it can resolve to different code over time. This creates a TOCTOU-style risk, as described in G07.

Impact and conditions. This is particularly dangerous in automated pipelines (*e.g.*, CI/CD or scheduled retraining) where models are periodically re-fetched, as a single malicious push to `main` can propagate to all consumers. Unlike G07, where the revision is simply absent, G08 involves an *explicit but mutable* reference, which may give a false sense of version control. The risk is mitigated only by pinning to an immutable commit hash or by using repository-level branch protection that prevents unauthorized pushes.

5.2.4 Use of Hard-coded Credentials (CWE-798). One security smell type (G09) relates to hardcoding authentication information [29].

G09 – Credential Exposure. It occurs when a configuration contains hard-coded credential material or fields strongly suggestive of credentials (*e.g.*, tokens, API keys, passwords, or secret-bearing identifiers). This can lead to sensitive data leakage because configuration files are often publicly visible, easy to clone, and widely reused. Any embedded secret may therefore be exposed to unauthorized users, enabling account compromise.

Impact and conditions. Exposed Hugging Face tokens can grant write access to model repositories, enabling an attacker to inject malicious artifacts at scale; leaked API keys for external services (*e.g.*,

cloud providers) can be abused for unauthorized usage or billed to the victim. The severity depends on the token's scope and permissions: a read-only token leaks less than a write-enabled one, but both constitute a breach. This issue can be worsened by the fact that configuration files are routinely copied across forks and mirrors, propagating the credential beyond their original context.

5.2.5 Sensitive Information in Log File (CWE-532). One security smell type (G10) relates to verbose or debug logging modes that may expose sensitive information during execution [27].

G10 – Debug Mode Enabled. When a configuration enables debugging, verbose logging, or similar diagnostic behavior, it can reveal unnecessary internal information in production settings (*e.g.*, file paths, runtime parameters, *etc.*). These can assist attackers in reconnaissance or leak sensitive operational context into logs.
Impact and conditions. In multi-tenant or API-exposed deployments, debug output can reveal model internals, user input data, or infrastructure details to unauthorized parties through log aggregation systems or error responses. In isolated development environments, the impact is limited to information leakage within the developer's own system.

5.2.6 Exposure of System Data (CWE-497). One smell type (G11) is related to the exposure of system-specific information [26].

G11 – Absolute Path Disclosure. When a configuration embeds an absolute filesystem path in fields such as `_name_or_path`, it can reveal details about the developer's or deployment environment's internal structure (similar to G10).
Impact and conditions. Leaked paths can reveal usernames, directory layouts, operating system details, and framework versions, lowering the cost of targeted follow-up attacks. The severity is typically low in isolation but increases when combined with other smells (*e.g.*, G09 credentials or G05 trust flags), as the environmental context aids exploit crafting.

5.2.7 Unbounded Resource Allocation (CWE-770). One security smell type (G12) is due to configuration files using excessively large or unbounded generation parameters [28].

G12 – Unbounded Generation Limits. When a configuration sets generation-related limits to extremely large values (*e.g.*, very high `model_max_length` or `max_new_tokens`), downstream systems may honor these values during inference, leading to excessive memory consumption, long-running requests, or denial-of-service conditions when exposed through APIs or interactive services. Thus, although the fields appear to be performance or quality settings, they can directly influence system availability.
Impact and conditions. In API-exposed settings, a single request honoring these limits can exhaust GPU memory or saturate compute, creating a low-effort denial-of-service vector. The risk depends on whether the serving infrastructure enforces its own resource bounds independently of the model configuration. Models intended for long-context use (*e.g.*, 128K-token models) may legitimately set high values, and the thresholds used to operationalize this smell are derived empirically from the corpus.

Table 1: Rules to detect security smells in model configuration files. *cfg*: parsed JSON; *own*: repo identity from file path.

Smell	Detection Rule
G01	$isDict(cfg[\texttt{auto\_map}]) \wedge \neg isCrossRepo(cfg, own) \wedge \neg isPEFT(cfg) \wedge \neg isTrust(cfg)$
G02	$\texttt{custom\_pipelines} \in cfg$
G03	$isDict(cfg[\texttt{auto\_map}]) \wedge \exists v \in cfg[\texttt{auto\_map}].\text{values}() : isCrossRepoRef(v) \wedge refRepo(v) \neq own$
G04	$isPEFT(cfg) \wedge \neg isTrust(cfg)$
G05	$cfg[\texttt{trust\_remote\_code}] = True$
G06	$\exists v \in vals(cfg, \text{depth} \leq 5) : isStr(v) \wedge isURL(v)$
G07	$isPEFT(cfg) \wedge \neg isTrust(cfg) \wedge \texttt{revision} \in cfg \wedge cfg[\texttt{revision}] = null$
G08	$\exists k \in cfg : \texttt{revision} \subseteq k \wedge isStr(cfg[k]) \wedge isMutableRev(cfg[k])$
G09	$\exists (k, v) \in cfg : isSecretField(k) \vee (isStr(v) \wedge isSecretValue(v))$
G10	$cfg[\texttt{debug}] \in \{True, 1\} \vee cfg[\texttt{verbose}] \in \{True, 1\} \vee cfg[\texttt{verbose\_logging}] \in \{True, 1\} \vee cfg[\texttt{log\_level}] \in \{\texttt{"DEBUG"}, \texttt{"debug"}\}$
G11	$isStr(cfg[\texttt{\_name\_or\_path}]) \wedge isAbsPath(cfg[\texttt{\_name\_or\_path}]) \wedge \neg isPEFT(cfg)$
G12	$\exists f \in \{\texttt{model\_max\_length}, \texttt{max\_seq\_len}, \texttt{max\_new\_tokens}, \texttt{max\_length}\} : isNum(cfg[f]) \wedge cfg[f] > threshold(f)$

$isPEFT \equiv \texttt{peft\_type} \in cfg$ $isTrust \equiv cfg[\texttt{trust\_remote\_code}]{=}True$ $refRepo(v)$ = 1st capture of $isCrossRepoRef$ $vals(cfg, d)$ = strings within depth d

5.3 Comparison to Prior Work

Compared to prior work on IaC security smells [41] and secret/misconfiguration scanners [4, 51, 53], our 12 smells fall into two categories: novel ML-specific security smells absent from existing taxonomies, and ML-specific manifestations of known security weaknesses that existing scanners fail to detect. First, G01–G06 and G12 have no counterpart in existing catalogs; detecting them requires ML-framework semantics: knowing that `auto_map` and `custom_pipelines` trigger dynamic class imports (G01–G03), that `peft_type` implies a transitive base-model load (G04), that `trust_remote_code` is a framework-specific safety guard (G05), that embedded URLs are fetched at initialization by the loader (G06), and that generation-limit fields govern inference-time resource allocation (G12). A generic JSON or IaC scanner cannot distinguish any of these from arbitrary metadata. Second, G07–G11 map to known weaknesses (CWE-494, CWE-798, CWE-532, CWE-497), yet none of the four scanners evaluated (§6.4) detect a single instance, because these smells manifest through ML-specific surfaces: `revision` fields whose relevance depends on co-present execution-enabling fields like `peft_type` (G07–G08), Hugging Face token prefixes in JSON metadata rather than source-code `.env` files (G09), debug flags honored by ML serving frameworks rather than web servers (G10), and absolute paths in ML-specific fields like `_name_or_path` outside existing path-disclosure rules (G11). Therefore, our taxonomy contributes *conceptual* novelty for the first group and *operational* novelty for the second, where existing tooling fails to detect ML-specific manifestations of known security weaknesses.

6 RQ2: Config. Security Smells Prevalence

To quantify the prevalence of model configuration security smells, we developed MONTICELLO[2], an approach that detects security smells in model configuration files.

6.1 MONTICELLO Overview

[2] Model lOading coNfiguraTIon seCurity smELLs detectOr

MONTICELLO operates directly on configuration artifacts (*e.g.*, *config.json*) and detects smells using a combination of *pattern-based predicates* and *rule-based composition*. As shown in Table 1, we define a set of reusable predicates (*e.g.*, *isURL*, *isMutableRev*, *isSecretValue*) and patterns that capture syntactic and semantic properties of configuration values. These predicates are implemented using regular expressions and value-set constraints, enabling lightweight yet expressive matching of risky constructs.

Thresholds for G12 were derived empirically from the corpus: the chosen values (10^6 for `model_max_length`/`max_seq_len`; 10^5 for `max_new_tokens`/`max_length`) sit at or above the 99.9th percentile of observed non-sentinel values (excluding placeholders such as $2^{31}-1$), flagging fewer than 0.3% of repositories in each case. The few borderline hits (*e.g.*, nine repos at `max_seq_len`=1,000,002) appear to be legitimate long-context declarations, reflecting a precision/recall tradeoff at the boundary rather than a clean cutoff.

Several smells, most notably G03, cannot be detected from the configuration file alone: determining whether an `auto_map` target points to a *different* repository requires knowing the identity of the current repository, which is derived from the file path or platform metadata, not from the JSON content. MONTICELLO extracts this identity (`own`) and passes it to predicates, such as *isCrossRepoRef* and *refRepo* (Table 1), to distinguish same-repo remapping (G01) from cross-repo trust-boundary expansion (G03), a discrimination that is invisible to tools operating on individual files in isolation.

Rule Specification. Each smell is defined as a logical rule that composes one or more predicates over configuration keys and values. Table 1 presents the full set of detection rules. For example, smells related to remote code execution (*e.g.*, `auto_map`, `trust_remote_code`) are identified by combining cross-repository reference detection (*isCrossRepoRef*) with execution-enabling fields, while credential exposure smells rely on both value-based patterns (*isSecretValue*) and field-name heuristics (*isSecretField*) [3]. This compositional design allows us to capture both direct security indicators and contextual signals that cannot be identified through isolated pattern matching.

Analysis Workflow. Given a configuration file, MONTICELLO parses it into key–value pairs and evaluates all predicates on candidate fields. The detection rules are then applied to determine whether a configuration exhibits one or more smells. The analysis is *intra-file* and does not require program execution, making it scalable to large corpora.

6.2 Detector Evaluation

6.2.1 Benchmarks. We built two benchmarks from real Hugging Face configuration files to evaluate our detector.

Microbenchmark (109 samples). The microbenchmark is derived from the **109 configuration instances** used in the inter-rater agreement study (Section 5.1.2). Each configuration was annotated as `TRUE` (genuine security concern) or `FALSE` (benign or mislabeled). We removed the **28 FALSE-labeled samples** and replaced them with manually verified true positives to ensure representation across all 12 canonical smell groups (at least two confirmed samples per group). As a result, the benchmark consists of 109 high-confidence true-positive samples spanning all smell groups. This dataset provides a *controlled evaluation setting* focused on detection capability under minimal label noise.

Minibenchmark (829 samples). This benchmark is created by sampling approximately **20 samples per initial 55 candidate smells identified in RQ1 (Section 5.1.2)**, resulting in 829 candidate configurations[4] that cover the full breadth of patterns observed during open coding. Each sample is then annotated against the *stabilized 12-smell taxonomy* (Figure 3): a single author assigned the primary smell label using the same codebook as the microbenchmark, and a second author independently verified a stratified 20% subset (at least two samples per smell group); all disagreements were resolved by the senior adjudicator (96.4% raw agreement on the verified subset). A sample whose label was contested during verification, or whose configuration exhibited ambiguous or co-occurring signals, was re-inspected jointly by both annotators and the adjudicator. Unlike the microbenchmark, this dataset preserves real-world characteristics such as overlapping signals, heterogeneous field usage, and co-occurrence of multiple smells, providing implicit negative examples for smell groups not present in a given sample.

Multi-label ground truth. As configurations often contain multiple co-occurring signals, we build multi-label variants by annotating the primary and all additional smell cases in each sample.

Benchmark coverage. The benchmark sizes reflect the cost of fully manual, multi-label annotation by domain experts; every label is verified by at least two authors, but the minibenchmark's 829 samples are drawn from all 55 initial candidate patterns, ensuring that the structural variations observed during open coding are represented. The two benchmarks are complementary by design: the microbenchmark tests detection against canonical, controlled instances of each smell, while the minibenchmark exposes the detector to in-the-wild heterogeneity including non-canonical encodings, co-occurring smells, and implicit negatives.

6.2.2 Evaluation Metrics. We evaluate each smell group independently, counting a true positive (TP) when a group appears in both the prediction and ground truth, a false positive (FP) when it appears only in the prediction, and a false negative (FN) when it appears only in the ground truth. This way, we compute *precision* (the fraction of correct predictions), *recall* (the fraction of ground-truth instances identified), and F1 (their harmonic mean).

6.3 Detector Results

Our approach achieves 92.2% F1 on the microbenchmark and 81.6% on the minibenchmark, indicating strong performance under both controlled and real-world settings (Table 2). High precision across both datasets shows most detections are valid, with performance differences primarily driven by recall under greater variability.

Near-perfect performance is observed for **G04**, **G06**, **G08**, and **G10**, which exhibit explicit and consistent patterns. Moderate performance for **G02** and **G03** reflects structural variability, while lower performance for **G01**, **G05**, and **G09** arises from non-canonical

[3] Due to space constraints, we omit some predicates (*e.g.*, isCrossRepo(c1,c2)); their detailed descriptions are in our replication package [46] (see file `predicate_reference.pdf`).

[4] 18 of the 55 smells had fewer than 20 samples

Table 2: MONTICELLO's Precision, Recall, and F1-Score %.

	Microbenchmark			Minibenchmark		
Smell	**Precision**	**Recall**	**F1-Score**	**Precision**	**Recall**	**F1-Score**
G01 AutoMap	100.0	73.8	84.9	100.0	40.2	57.3
G02 Pipeline	100.0	81.8	90.0	100.0	44.1	61.2
G03 Cross-Repo	100.0	91.3	95.5	100.0	87.4	93.3
G04 PEFT	100.0	100.0	100.0	100.0	93.3	96.5
G05 Trust	100.0	50.0	66.7	100.0	19.0	32.0
G06 External URL	100.0	100.0	100.0	100.0	100.0	100.0
G07 Revision Null	100.0	94.1	97.0	100.0	99.7	99.8
G08 Mutable Rev	100.0	100.0	100.0	100.0	100.0	100.0
G09 Credential	100.0	92.3	96.0	100.0	86.3	92.6
G10 Debug	100.0	100.0	100.0	100.0	100.0	100.0
G11 Abs Path	100.0	100.0	100.0	100.0	42.4	59.6
G12 Gen Limits	100.0	100.0	100.0	100.0	66.7	80.0
Overall	100.0	85.5	92.2	100.0	68.9	81.6

Table 3: Smell prevalence per platform.

	Hugging Face		ModelScope		OpenCSG	
Smell	**# Repos**	**% Repos**	**# Repos**	**% Repos**	**# Repos**	**% Repos**
G01 AutoMap	13,230	36.5	2,871	90.6	2,170	40.0
G02 Pipeline	354	1.0	1	0.0	167	3.1
G03 Cross-Repo	8,317	23.0	63	2.0	432	8.0
G04 PEFT	5,254	14.5	1	0.0	4	0.1
G05 Trust	133	0.4	26	0.8	7	0.1
G06 External URL	16	0.0	8	0.3	3	0.1
G07 Revision Null	5,240	14.5	1	0.0	4	0.1
G08 Mutable Rev	14	0.0	0	0.0	1	0.0
G09 Credential	5,768	15.9	1,579	49.8	1,325	24.4
G10 Debug	3	0.0	0	0.0	0	0.0
G11 Abs Path	2,490	6.9	413	13.0	731	13.5
G12 Gen Limits	387	1.1	332	10.5	94	1.7
Any smell	28,510	78.7	3,028	95.5	3,554	65.5

`auto_map` patterns, implicit or context-dependent indicators (*e.g.*, omitted flags or heterogeneous encodings).

On one hand, moderate performance is observed for **G01** and **G02**, reflecting variability in how these constructs are specified across repositories and the presence of multiple valid patterns that do not always share identical structural signatures. Lower performance persists for **G05** and **G09**. For **G05**, the relevant configuration indicator (*e.g.*, the presence or absence of `trust_remote_code`) is often implicit or omitted, leading to reduced recall. For **G09**, detection is challenged by heterogeneous value encodings and context-dependent patterns, which require reasoning beyond simple syntactic matching.

Limitations. While effective for identifying syntactic and structural risks, Monticello relies on rule-based patterns and does not perform deep semantic or runtime analysis. As such, residual false negatives arise from structural variability and incomplete manifestation of relevant configuration indicators, where equivalent behaviors are expressed using non-canonical patterns (notably in **G01** and **G11**). Moreover, some smells depend on implicit or context-dependent indicators (*e.g.*, omitted flags or dynamically resolved values), which are difficult to capture solely through static analysis. These limitations reflect an inherent trade-off between precision and recall in rule-based approaches and highlight the need for deeper semantic analysis to improve coverage.

6.4 Baseline Comparison

Prior work on IaC security (*e.g.*, SLIC [41]) targets Puppet and Chef scripts and does not generalize to JSON-based configuration artifacts used in model repositories. We also explored ConfigScan [11], an LLM-based configuration analyzer, but it is not publicly available. Thus, we compare our work against widely used, off-the-shelf security scanners: *detect-secrets* [53], *gitleaks* [51], *Checkov* [4], and *Semgrep*'s built-in `p/secrets` ruleset [44]. These scanners are industry-standard approaches for detecting security issues in code and configuration artifacts, with a particular focus on hard-coded secrets, credential leakage, and infrastructure misconfigurations.

All scanners had *zero detections* on the microbenchmark. This is unsurprising as these tools are designed to identify traditional secrets or infrastructure misconfigurations and do not model execution-relevant configuration semantics specific to ML systems. In particular, they lack support for core smell groups such as **G01 (AutoMap)**, **G03 (Cross-Repository Code Reference)**, **G04 (PEFT Adapter)**, and **G12 (Generation Limits)**, which capture execution behavior rather than explicit vulnerabilities. In contrast, Monticello had 100% precision, 96.1% recall, and 98% F1. These results highlight two findings: (1) ML configuration smells constitute a distinct class of security risks that are not captured by existing tools; (2) incorporating repository-aware reasoning enables more accurate detection, particularly in the presence of co-occurring smells.

We do not report paired statistical tests (*e.g.*, McNemar's [38]) for the baseline comparison because all four scanners produced zero detections across all smell groups, leaving no discordance to test; the comparison is categorical rather than quantitative.

6.5 Smells Occurrence in Model Hubs

As shown in Table 3, configuration smells are pervasive across all platforms, affecting **78.7%** of Hugging Face, **95.5%** of ModelScope, and **65.5%** of OpenCSG repositories. **G01** is the most prevalent execution-enabling smell, impacting **36.5%** of Hugging Face, **90.6%** of ModelScope, and **40.0%** of OpenCSG repositories. **G03** (Cross-Repository AutoMap) appears in **23.0%** of Hugging Face repositories, indicating substantial cross-repository trust expansion. **G09** is common across all platforms (**15.9–49.8%**), suggesting frequent presence of credential-like patterns. **G04** and **G07** each appear in ∼14.5% of Hugging Face repositories but are rare on ModelScope and OpenCSG, reflecting Hugging Face's larger PEFT adapter ecosystem. **G11** affects **6.9–13.5%** of repositories, while **G12** appears in **1.1–10.5%**. Other smells, including **G05**, **G06**, **G02**, and **G08**, occur infrequently (generally below 3%), but represent high-impact risks when present. Lastly, we observe that ModelScope exhibits the highest incidence of execution-related smells (90.6% for G01) and credential-related smells (49.8% for G09), reflecting a highly dynamic, extensible model-loading ecosystem. Hugging Face shows moderate G01 prevalence (36.5%) but the highest rate of cross-repository references (G03, 23.0%) and PEFT adapters (G04, 14.5%). In contrast, OpenCSG shows the lowest overall smell rate (65.5%) with a more balanced distribution across smell groups.

To test whether prevalence differences across platforms reflect genuine effects, we ran Pearson's χ^2 test of independence [37] over the full 12×3 (smell $\times$ platform) contingency table from Table 3 and report Cramér's V [10] as effect size (Table 4). Per-smell tests are significant for 10 of 12 groups, with **G01 (AutoMap)** showing the largest effect (V=0.281, medium), followed by **G09** (V=0.224) and **G12** (V=0.180). The two rarest smells: **G08** and **G10**,violate the minimum-expected-count assumption; pairwise Fisher's exact

Table 4: Platform association per smell (dof=2) with Cramér's V [10]. †: Fisher's exact [12] used (Holm–Bonferroni corrected [16]).

	G01	G02	G03	G04	G05	G06†	G07	G08†	G09	G10†	G11	G12
χ^2	3526.59	219.17	1346.00	1399.42	25.94	21.01	1395.16	1.72	2246.78	0.71	387.07	1447.56
p	<.001	<.001	<.001	<.001	<.001	<.001	<.001	.424	<.001	.701	<.001	<.001
V	.281	.070	.173	.177	.024	.022	.176	.006	.224	.004	.093	.180
Sig.	***	***	***	***	***	***	***	ns	***	ns	***	***

*** $p < .001$; ns = not significant (α=.05).

tests [12] with Holm–Bonferroni correction [16] find no significant platform association for either, indicating their rarity is uniform across platforms. We also examined whether popular repositories exhibit fewer smells by computing Spearman correlations [50] between download count and smell presence. Overall smell density shows negligible correlation with popularity ($|\rho| \leq 0.094$), but **G01** is positively correlated with downloads on Hugging Face (ρ=+0.452, $p < .001$, medium effect), indicating that widely consumed models, which increasingly adopt composite architectures requiring `auto_map`, carry *more*, not fewer, execution-enabling constructs.

7 RQ3: Bridge-Referenced Code Security

Model loading configuration files containing any of the configuration smells G01, G02, and G03 would cause the model loading framework to import and execute Python code residing in the same repository as the configuration file. Beyond the configuration-level risks previously discussed, the *referenced Python code itself* may contain additional vulnerabilities and security smells. To investigate this possibility, we perform a static security analysis of the Python files loaded transitively via these bridge fields.

7.1 Study Methodology

For each configuration flagged with G01, G02, or G03, we extract the Python module paths referenced in the `auto_map` and `custom_pipelines` fields. Each entry follows the format `org/repo--module.ClassName`. We map these references to the corresponding source files within the repository and include them in the analysis set, resulting in 21,474 Python files for Hugging Face, 5,345 for ModelScope, and 4,636 for OpenCSG. Next, we apply three complementary static analyzers to the identified Python files to detect security smells. We use **Bandit** [39], a Python security linter that identifies common issues (*e.g.*, command injection, insecure deserialization, weak cryptography, hardcoded credentials, and unsafe system calls); **Semgrep** [44], a pattern-based analyzer with Python security rules targeting vulnerabilities such as CWE-78 (OS command injection), CWE-502 (unsafe deserialization), and CWE-798 (hardcoded credentials); and **CodeQL** [13], a semantic analysis engine that leverages data and control flow to detect deeper vulnerabilities, including injection flaws, unsafe deserialization, and insecure library usage.

We chose these analyzers because they are widely adopted, employed as standard baselines for large-scale vulnerability analysis, actively maintained, and collectively cover a broad spectrum of security issues, as per previous studies [3, 14, 20, 22, 45, 48, 49]. Bandit provides lightweight detection of common Python-specific anti-patterns, Semgrep enables flexible detection of CWE-aligned patterns, and CodeQL captures deeper semantic vulnerabilities through inter-procedural analysis.

7.2 Results

Table 5 reports affected repositories, total findings, and dominant vulnerability categories. Across all platforms, **Bandit** findings are dominated by **CWE-703 (Improper Exception Handling)**, accounting for **76–86%** of issues. This is largely driven by **B101–`assert` usage** (**73–84%** of findings) indicating widespread reliance on fragile error-handling practices. The primary medium-severity issue is **B615**, highlighting unsafe remote downloads without integrity checks. This pattern is consistent across Hugging Face, OpenCSG, and ModelScope, suggesting systemic risks in exception handling and dependency retrieval during model loading.

Semgrep findings are dominated by **CWE-502 (Untrusted Deserialization)** and **CWE-95 (Eval Injection)**. These results indicate that model-loading pipelines often rely on dynamic artifacts execution. Unlike Semgrep, **CodeQL** findings are heavily concentrated in **timing side-channel vulnerabilities**, with **96–99%** of detections corresponding to "Timing attack against secret." These issues arise from non-constant-time comparisons in security-sensitive contexts. Other categories, such as cryptographic misuse and regex vulnerabilities, appear at much lower frequencies. Overall, CodeQL highlights risks in authentication and input validation logic.

8 Discussion

AI Supply Chain and Transitive Risks. Modern model hubs effectively operate as AI supply chains, where configuration artifacts control not only local execution but also the retrieval and composition of remote dependencies. Fields such as `auto_map` and adapter configurations introduce transitive execution paths beyond the originating repository. Unlike traditional package ecosystems, these dependencies lack safeguards such as version pinning or integrity checks, exposing users to supply-chain risks, including dependency confusion and malicious updates. Configuration artifacts, therefore, act as *supply-chain amplifiers*, determining which code is executed and under what trust assumptions.

Stealth Trust-Boundary Expansion. We observe that configuration artifacts stealthily influence execution prior to or independently of explicit safety controls (*e.g.*, `trust_remote_code`). This creates a mismatch between user expectations and actual execution behavior, as configurations can implicitly import modules, fetch dependencies, and activate dynamic code paths. Since these behaviors are neither explicitly declared nor surfaced to users, configuration files function as a stealth control channel that can bypass both developer intent and platform safeguards.

Limitations of Existing Security Tools. Our evaluation shows that existing security tools are ineffective in this setting as they rely on syntactic patterns, whereas ML configurations encode execution behavior through domain-specific constructs (*e.g.*, `auto_map`, `peft_type`). Effective detection requires reasoning across multiple fields, repository context, and framework-specific semantics, which existing approaches do not support.

Implications for Platforms, Tooling, and Developers. Our findings have several implications for secure model-sharing ecosystems.

Table 5: Repository-level prevalence of security issues identified by Bandit, Semgrep, and CodeQL across three model hubs. Bandit severity: H (high impact), M (moderate risk), L (low-risk / best-practice issues).

	Bandit					Semgrep					CodeQL				
	#Repos	#Issues		Results		#Repos	#Issues		Results		#Repos	#Issues		Results	
Hugging Face	6,154	35,475	Top 3 CWEs	CWE-703 Exception Handling	29,072 82.0%	1,267	1,982	Top 3 CWEs	CWE-502 Deserialization	704 35.5%	2,571	5,855	Top 3 CWEs	CWE-208 Timing Discrepancy	5,693 97.2%
				CWE-494 No Integrity Check	3,126 8.8%				CWE-95 Eval Injection	397 20.0%				CWE-327 Weak Cryptography	120 2.0%
				CWE-259 Hard-coded Pass	1,735 4.9%				CWE-706 Ref. Resolution	16 0.8%				CWE-020 Input Validation	30 0.5%
			Top 1 Sev.	H: None	— —			Top 3 Rules	numpy-modules	812 41.0%			Top Rules	Timing attack	5,688 97.2%
				M: Unsafe download	3,126 8.8%				pickle usage	649 32.7%				Crypto issues	50 0.9%
				L: Use of assert	28,239 79.6%				eval detected	397 20.0%				Hash issues	50 0.9%
OpenCSG	1,407	7,293	Top 3 CWEs	CWE-703 Exception Handling	5,539 76.0%	201	331	Top 3 CWEs	CWE-502 Deserialization	134 40.5%	376	1,065	Top 3 CWEs	CWE-208 Timing Discrepancy	1,033 97.0%
				CWE-494 Integrity Check	799 11.0%				CWE-95 Eval Injection	76 23.0%				CWE-327 Weak Cryptography	25 2.3%
				CWE-259 Hard-coded Pass	653 9.0%				CWE-706 Ref. Resolution	10 3.0%				CWE-020 Input Validation	7 0.7%
			Top 1 Sev.	H: None	— —			Top 3 Rules	pickle usage	134 40.5%			Top Rules	Timing attack	1,029 96.7%
				M: Unsafe download	799 11.0%				numpy-modules	105 31.7%				Crypto	10 0.9%
				L: Use of assert	5,329 73.1%				eval	76 23.0%				Hash	10 0.9%
ModelScope	1,737	11,087	Top 3 CWEs	CWE-703 Exception Handling	9,498 85.7%	400	478	Top 3 CWEs	CWE-502 Deserialization	159 33.3%	1,078	2,155	Top 3 CWEs	CWE-208 Timing Discrepancy	2,131 98.9%
				CWE-259 Hard-coded Pass	925 8.3%				CWE-95 Eval Injection	37 7.7%				CWE-020 Input Validation	16 0.7%
				CWE-494 Integrity Check	405 3.7%				CWE-96 Code Injection	6 1.3%				CWE-327 Weak Cryptography	5 0.2%
			Top 1 Sev.	H: None	— —			Top 3 Rules	numpy-modules	270 56.5%			Top Rules	Timing attack	2,131 98.9%
				M: Unsafe download	405 3.7%				pickle usage	155 32.4%				Regex issue	16 0.7%
				L: Use of assert	9,281 83.7%				eval	37 7.7%				HTML filter	3 0.1%

First, **Platforms** should enforce smell checks during configuration parsing at upload or pull time. For example, consider a repository whose `config.json` contains an `auto_map` entry pointing to a different organization (*e.g.*, `"AutoModelForCausalLM":"evil-org/payload--model.Exploit"`). Monticello flags this as both G01 (Auto Class Remapping, §5.2.1) and G03 (Cross-Repository AutoMap, §5.2.2); a platform-side check at upload time could reject or quarantine the repository and notify the uploader, blocking the attack at the supply-chain entry point using the same `isCrossRepoRef` predicate Monticello employs. Platforms should likewise enforce version pinning for external references (G07/G08, §5.2.3–5.2.3) and scan for embedded credentials (G09, §5.2.4) before publication. Second, **Tools** must incorporate ML-specific configuration semantics and repository context; our baseline evaluation (§6.4) shows that none of the four industry-standard scanners detect any of the twelve smell types. Integrating Monticello's predicate-based checks into CI/CD pipelines or platform-side hooks would require minimal effort, since the analysis is intra-file, execution-free, and runs in seconds per repository. Third, **Developers** should treat configuration files as executable specifications: auditing `auto_map` and `trust_remote_code` changes as rigorously as source code, pinning external references to immutable commit hashes rather than mutable branches like `main` (G08, §5.2.3), and sanitizing environment-specific fields such as `_name_or_path` before publication (G11, §5.2.6). Overall, securing modern ML ecosystems requires moving beyond code-centric models to jointly analyze configurations, dependencies, and execution flows.

8.1 Threats to Validity

Construct validity. We mitigate threats to our manual taxonomy construction by performing structured open coding, followed by multi-rater validation and adjudication using a shared codebook. Furthermore, our tool is developed based on observable patterns identified during this process. However, some heuristic patterns (*e.g.*, credential-like fields) may introduce false positives, which we mitigate through rule refinement, contextual constraints (*e.g.*, field-name filtering), and validation on curated benchmarks.

Internal validity. Monticello combines rule-based detection with repo-aware analysis, which may introduce errors due to rule expressiveness, parsing limitations, *etc.*. We mitigate this by performing a multi-label evaluation to account for co-occurring smells.

External validity. We study three major model-sharing platforms, Hugging Face, ModelScope, and OpenCSG, covering a large portion of the open model ecosystem. While results may not generalize to more controlled environments (*e.g.*, NVIDIA NGC), these platforms capture prevalent real-world practices, making them suitable for large-scale analysis. Our analysis uses Bandit, Semgrep, and CodeQL, widely adopted and complementary tools covering syntactic and semantic vulnerabilities [3, 14, 20, 22, 45, 47–49]. We mitigate individual limitations by combining analyzers and focusing on consistent cross-tool patterns. This setup reflects best practices in large-scale security analysis and provides a robust, reproducible approximation of real-world risk.

Conclusion validity. Results depend on benchmark construction and sampling. We mitigate this by combining controlled (micro) and realistic (mini) datasets derived from real configurations and manually annotated, covering both canonical and in-the-wild patterns to aggregate trends across datasets.

9 Conclusion

We present an empirical study of *Execution-as-Configuration* in model-sharing ecosystems. We introduce a taxonomy of 12 configuration security smells and, guided by this taxonomy, analyze 44,812 repositories across three platforms. This analysis reveals that configuration-driven risks are pervasive, with a small number of constructs (*e.g.*, *auto_map*) dominating the attack surface. We also show that these smells translate into security risks (*e.g.*, RCE) and that existing security tools are ineffective in this setting.

10 Data Availability

Our replication package is publicly available at [46].

References

[1] 2026. CVE-2026-45829: Pre-Authentication Code Injection in ChromaDB via Malicious Model Configuration. https://nvd.nist.gov/vuln/detail/CVE-2026-45829.

[2] 2026. CVE-2026-5241: LightGlue Configuration Propagates `trust_remote_code` from `config.json` into Nested Model Loading in HuggingFace Transformers. https://nvd.nist.gov/vuln/detail/CVE-2026-5241.

[3] 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. *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 614–623. doi:10.1145/3661167.3661262

[4] Bridgecrew, Inc. 2024. Checkov: Prevent cloud misconfigurations and find vulnerabilities during build-time in infrastructure as code. https://github.com/bridgecrewio/checkov.

[5] 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. arXiv:2005.14165 http://arxiv.org/abs/2005.14165v4

[6] Beatrice Casey, Kaia Damian, Andrew Cotaj, and Joanna C. S. Santos. 2025. An Empirical Study of Safetensors' Usage Trends and Developers' Perceptions. arXiv:2501.02170 [cs.SE] https://arxiv.org/abs/2501.02170

[7] Beatrice Casey, Joanna C. S. Santos, and Mehdi Mirakhorli. 2024. A Large-Scale Exploit Instrumentation Study of AI/ML Supply Chain Attacks in Hugging Face Models. arXiv:2410.04490 [cs.CR] https://arxiv.org/abs/2410.04490

[8] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[9] Juliet M Corbin and Anselm Strauss. 1990. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology* 13, 1 (1990), 3–21. doi:10.1007/BF00988593

[10] Harald Cramér. 1999. *Mathematical methods of statistics*. Vol. 9. Princeton university press.

[11] Ziqi Ding, Qian Fu, Junchen Ding, Gelei Deng, Yi Liu, and Yuekang Li. 2025. A Rusty Link in the AI Supply Chain: Detecting Evil Configurations in Model Repositories. In *2025 IEEE Security and Privacy Workshops (SPW)*. IEEE, 260–264. doi:10.1109/SPW67851.2025.00036

[12] Ronald A Fisher. 1922. On the interpretation of χ 2 from contingency tables, and the calculation of P. *Journal of the royal statistical society* 85, 1 (1922), 87–94. doi:10.2307/2340521

[13] GitHub Security Lab. 2023. CodeQL: Semantic Code Analysis Engine. https://codeql.github.com/. Accessed: 2026.

[14] Damian Gnieciak and Tomasz Szandala. 2025. Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection. 198410-198422 pages. arXiv:2508.04448 [cs.SE] doi:10.1109/access.2025.3635168

[15] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (Oct. 1988), 36–38. doi:10.1145/54289.871709

[16] Sture Holm. 1979. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70. http://www.jstor.org/stable/4615733

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

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

[19] Wenxin Jiang, Nicholas Synovic, Rohan Sethi, Aryan Indarapu, Matt Hyatt, Taylor R Schorlemmer, George K 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*. 105–114. doi:10.1145/3560835.3564547

[20] 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] doi:10.48550/ARXIV.2509.02022

[21] A. Kellas, Neophytos Christou, Wenxin Jiang, Penghui Li, Laurent Simon, Yaniv David, V. Kemerlis, James C. Davis, and Junfeng Yang. 2025. PickleBall: Secure Deserialization of Pickle-based Machine Learning Models. https://arxiv.org/abs/2508.15987. doi:10.1145/3719027.3765037 arXiv:2508.15987.

[22] 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. *Lecture Notes in Computer Science*, 64–83. doi:10.1007/978-3-031-64171-8_4

[23] Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications. doi:10.4135/9781071878781

[24] MITRE. 2025. CWE-470: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection'). https://cwe.mitre.org/data/definitions/470.html.

[25] MITRE. 2025. CWE-494: Download of Code Without Integrity Check. https://cwe.mitre.org/data/definitions/494.html.

[26] MITRE. 2025. CWE-497: Exposure of System Data. https://cwe.mitre.org/data/definitions/497.html.

[27] MITRE. 2025. CWE-532: Insertion of Sensitive Information into Log File. https://cwe.mitre.org/data/definitions/532.html.

[28] MITRE. 2025. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

[29] MITRE. 2025. CWE-798: Use of Hard-coded Credentials. https://cwe.mitre.org/data/definitions/798.html.

[30] MITRE. 2025. CWE-829: Inclusion of Functionality from Untrusted Control Sphere. https://cwe.mitre.org/data/definitions/829.html.

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

[32] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. 2019. Ludwig: a type-based declarative deep learning toolbox. *arXiv preprint arXiv:1909.07930* (2019). doi:10.48550/arXiv.1909.07930

[33] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings* (Lisbon, Portugal). Springer-Verlag, Berlin, Heidelberg, 23–43. doi:10.1007/978-3-030-52683-2_2

[34] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

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

[36] Long Ouyang, Jeffrey Wu 0003, 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 F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *NeurIPS* (2022). https://dblp.org/rec/conf/nips/Ouyang0JAWMZASR22

[37] Karl Pearson. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (1900), 157–175. doi:10.1080/14786440009463897

[38] Matilda QR Pembury Smith and Graeme D Ruxton. 2020. Effective use of the McNemar test. *Behavioral Ecology and Sociobiology* 74, 11 (2020), 133. doi:10.1007/s00265-020-02916-y

[39] PyCQA. 2014. Bandit: A Security Linter for Python. https://bandit.readthedocs.io. Accessed 2026.

[40] PyTorch. 2025. torch.hub — PyTorch Documentation. https://docs.pytorch.org/docs/stable/hub.html Accessed: 2025-10-13.

[41] 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, 164–175. doi:10.1109/ICSE.2019.00033

[42] Akond Rahman, Asif Partho, Patrick Morrison, and Laurie Williams. 2018. What questions do programmers ask about configuration as code?. In *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*. 16–22. doi:10.1145/3194760.3194769

[43] Joanna CS Santos. 2025. Cyber-AI Supply Chain Vulnerabilities. *AI for Cybersecurity: Research and Practice* (2025), 451–470. doi:10.1002/9781394293773.ch16

[44] Semgrep, Inc. 2024. Semgrep: Static Analysis Tool for Finding Bugs, Security Issues, and Code Smells. https://semgrep.dev. Accessed: 2026-03-26.

[45] Mohammed Latif Siddiq, Joanna Cecilia da Silva Santos, Sajith Devareddy, and Anna Muller. 2024. 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). *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 54–65. doi:10.1145/3691621.3694934

[46] Mohammed Latif Siddiq, Prince Noah Johnson, and Joanna Cecilia da Silva Santos. 2026. *Execution-as-Configuration: Security Smells in Model Configuration Artifacts*. doi:10.5281/zenodo.19340072

[47] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS 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)*. IEEE, 71–82. doi:10.1109/SCAM55253.2022.00014

[48] Mohammed Latif Siddiq, Tanzim Hossain Romel, Natalie Sekerak, Beatrice Casey, and Joanna C. S. Santos. 2026. An Empirical Study on Remote Code Execution in Machine Learning Model Hosting Ecosystems. arXiv:2601.14163 [cs.SE] https://arxiv.org/abs/2601.14163

[49] 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). *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, 29–33. doi:10.1145/3549035.3561184

[50] Charles Spearman. 1961. The proof and measurement of association between two things. (1961). doi:10.2307/1412159

[51] Zachary Toews. 2024. Gitleaks: Protect and discover secrets using Gitleaks. https://github.com/gitleaks/gitleaks.

[52] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi 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 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*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, Online, 38–45. doi:10.18653/v1/2020.emnlp-demos.6

[53] Yelp, Inc. 2023. detect-secrets: An enterprise friendly way of detecting and preventing secrets in code. https://github.com/Yelp/detect-secrets.

[54] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45. https://api.semanticscholar.org/CorpusID:83459546

[55] 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). *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2087–2098. doi:10.1145/3691620.3695271

[56] 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*. 2087–2098. doi:10.1145/3691620.3695271

[57] Peng Zhou. 2024. How to Make Hugging Face to Hug Worms: Discovering and Exploiting Unsafe Pickle.loads over Pre-Trained Large Model Hubs. https://www.blackhat.com/asia-24/briefings.html.

[58] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security symposium (USENIX security 19)*. 995–1010.

Received 2026-03-26; accepted 2026-06-18