BOLO: A VERIFIED MODEL HUB WITH READY-TO-RUN INFERENCE PROGRAMS FOR AGENTS AND HUMANS

Anonymous authors
Paper under double-blind review

ABSTRACT

Both AI agents and human engineers can greatly benefit from the ability to use any public machine learning model. Unfortunately, most public models, including those on Hugging Face, are not usable out of the box, and manually writing inference programs does not scale because failures follow diverse, model-specific patterns. Even worse, existing coding agents often produce faulty or misleading inference programs that run successfully yet silently behave differently than intended. Addressing this requires more sophisticated, model-specific verification strategy that can identify the ways those generated programs deviate from user intentions. To this end, we introduce BOLO, a large-scale model hub for constructing and verifying model inference programs. BOLO verifies each generated program at two levels: behavioral checks that test whether outputs are valid for the model's task, and content-level verification that checks the implementation. Across five model-dependent tasks, agents using BOLO achieve higher-quality outputs than agents with direct Hugging Face access while reducing LLM cost by up to 26%. On 8,570 most widely downloaded Hugging Face models, BOLO covers 65.2%, compared with 41.6% for Hugging Face Transformers `pipeline` API, including 1,195 models outside its support. Finally, among 2,019 programs passing behavioral checks, BOLO's content-level verification judges 88.6% clean, compared with 57.2% for a binary LLM judge. Qualitative case studies further demonstrate that BOLO avoids unsupported rejections while identifying genuine implementation issues that output checks alone cannot detect.

1 INTRODUCTION

Providing an easy-to-use mechanism for the proper use of machine learning models benefits both humans and AI agents. For human users, utilizing off-the-shelf models eliminates the need to invest the data, computational power, and expertise required to train models from scratch (Bommasani et al., 2022; Wolf et al., 2020); it also lowers the barrier for non-machine learning experts to apply state-of-the-art technologies in their work (Abid et al., 2019). For AI agents, invoking external models can endow general-purpose language models with capabilities they otherwise lack. For instance, systems such as HuggingGPT and Visual ChatGPT address multi-modal tasks by orchestrating specialized models from domains including vision and speech (Shen et al., 2023; Li et al., 2023; Lu et al., 2023; Liang et al., 2023; Wu et al., 2023).

However, although model hubs such as Hugging Face[1] host millions of public models spanning various domains, neither these hubs nor the systems built on top of them allow for using any of the public models correctly and effortlessly. A model repository may contain weights and documentation without a ready-to-run inference program, while standardized interfaces such as the Hugging Face Transformers (`pipeline` API) cover only models already integrated into their implementations, and even some of these integrations fail in practice, as shown in Section 3.2 (Wolf et al., 2020). Model-augmented agents (Shen et al., 2023; Lu et al., 2023; Li et al., 2023) therefore typically assume that candidate models have already been exposed as executable tools, confining them to a fixed, pre-integrated subset whose coverage cannot scale with the hub. Accessing the remaining models requires agents or users to interpret model-specific instructions, resolve dependencies, and

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

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

repair code. Consequently, the breadth of a model hub does not translate into broad, unified, and ready-to-use model access.

We make a key observation: *the critical component missing in the community is the ability to create verified, easy-to-use inference programs in a scalable way.* Here, a verified inference program refers to a model-specific executable interface that users and agents can invoke directly, accompanied by evidence of how faithfully it implements the model's intended inference semantics. Constructing such programs at scale is challenging because models vary in architectures, dependencies, and implementations. Verifying them is equally difficult. Their outputs can be stochastic (e.g., sampled tokens from a language model). A program may execute successfully yet contain *deviations*, i.e., discrepancies from the model's intended inference semantics, such as internally routing inputs to another model. Moreover, whether a deviation matters depends on user expectations, so a binary verdict cannot adequately capture it: issuing one requires fixing in advance which deviations make a program incorrect, yet this distinction is inherently ambiguous and cannot be enumerated across models, tasks, and use cases. The verdict thus reflects this predefined boundary as much as the program itself. A scalable solution must therefore adapt to individual models and report deviations explicitly rather than collapsing every finding into a binary usability decision.

We introduce BOLO, a verified model hub that constructs and verifies model-specific inference programs at scale for Hugging Face models. BOLO's key novelty is verifying these programs at scale, checking both what a program outputs and how it produces them. Rather than issuing a simple usability label, Bolo presents these deviations alongside each program, enabling agents to invoke models directly and users to assess whether an implementation meets their expectations.

Our evaluation shows that BOLO improves agent capability and efficiency, broadens model inference access, and makes verification more informative:

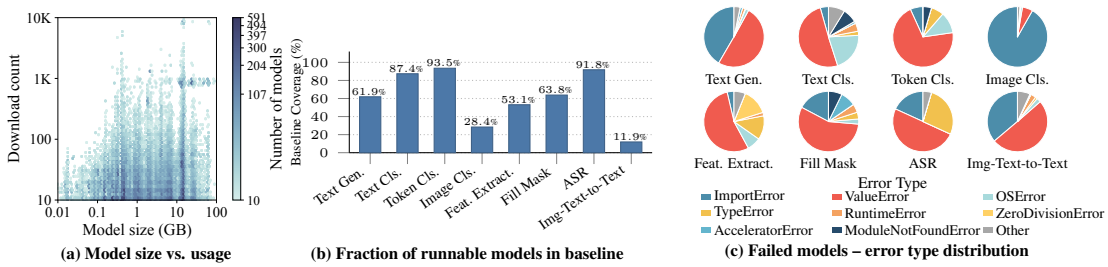
- **Agents accomplish more at lower cost.** Agents that access specialized models through BOLO produce higher-quality outputs than agents with direct Hugging Face access, while reducing total LLM cost by up to 26%.
- **Far more models become usable.** BOLO makes 65.2% of the evaluated models executable through a single interface, compared with 41.6% for the Transformers `pipeline` API, including 1,195 models that `pipeline` does not support at all.
- **Verification tells users more than pass/fail.** Among programs that pass behavioral checks, BOLO finds no deviations in 88.6%, whereas a binary verdict clears only 57.2%. At the same time, BOLO surfaces substantive deviations that behavioral testing misses.

2 RELATED WORK

Model-augmented agents. Existing model-augmented agents demonstrate the value of specialized models, but generally assume that these models are already available as executable and reliable tools. HuggingGPT, Visual ChatGPT, and ModelScopeGPT coordinate collections of expert models to extend LLMs across tasks and modalities (Shen et al., 2023; Wu et al., 2023; Li et al., 2023). TaskMatrix.AI and Chameleon similarly compose external models and APIs to solve complex tasks (Liang et al., 2023; Lu et al., 2023). These works primarily study how agents plan, select, and invoke tools on available model interfaces. In contrast, BOLO addresses the preceding infrastructure challenge: it turns heterogeneous models into verified tools that agents can use seamlessly.

Unified model inference interfaces. Existing unified inference interfaces depend on manual integration and provide no per-model verification. Hugging Face Transformers and ModelScope expose models through task-level pipelines (Wolf et al., 2020; Team, 2023), but only after each architecture has been manually implemented in the library and registered to a supported task. As a result, many hosted models remain unreachable. Moreover, because integration happens at the architecture level, a pipeline may run on an individual model with errors (see Section 3.2). BOLO addresses both limitations by automatically constructing model-specific inference programs, verifying each one, and exposing them through a common interface.

LLM-based program verification. Existing program-verification methods define correctness through expected outputs, known defect classes, formal properties, or holistic judgments, none of which captures whether an inference program faithfully realizes a model's intended inference semantics. Test-based methods such as CodeT and EvalPlus generate test cases to select or reject



(a) Model size vs. usage **(b) Fraction of runnable models in baseline** **(c) Failed models – error type distribution**

Figure 1: We analyze Hugging Face models to understand (a) the relationship between model size and usage, (b) the fraction of models runnable through the baseline `pipeline` interface, and (c) the error-type distribution among failed models. The scale of the model collection and the diversity of failures motivate an automated approach to constructing a broadly accessible model hub.

candidate solutions (Chen et al., 2022; Liu et al., 2023), auditing methods such as IRIS and RepoAudit combine LLMs with static analysis or path reasoning to detect vulnerabilities and bugs (Li et al., 2025; Guo et al., 2025), formal methods pair LLMs with theorem provers to prove specified properties (Lin et al., 2024), and LLM-as-a-judge approaches assess correctness against a problem specification (Wang et al., 2026). Model inference programs, however, often admit multiple valid outputs, lack formal specifications, and can execute successfully while deviating from the intended inference semantics. BOLO therefore combines behavioral and content-level verification and reports localized deviations rather than a single pass/fail verdict.

3 MOTIVATION

We motivate BOLO by showing that model access expands agent capabilities, while broad, dependable access requires model-specific inference programs and verification beyond successful execution and output checks.

3.1 MODEL ACCESS EXPANDS AGENT CAPABILITIES

We use a case study to show that an agent solves a challenging task far more successfully when it can run pretrained models from a model hub such as Hugging Face, yet raw hub access leaves it substantial integration work.

Case study. We ask Mini-SWE-agent (Yang et al., 2024) to separate ten two-speaker speech mixtures and transcribe each speaker. This task requires both speech separation and speech recognition. We measure separation by the scale-invariant signal-to-distortion ratio improvement (SI-SDRi) over the input mixture (Roux et al., 2018) and transcription by word error rate (WER); higher SI-SDRi and lower WER are better. With no hub access, the agent combines non-negative matrix factorization and PocketSphinx (Huggins-Daines et al., 2006) into a pipeline that runs without error but produces low-quality output. The separated streams are worse than the input mixture (-2.22 dB SI-SDRi) and the transcripts are unusable (WER 1.00). With Hugging Face access, the agent separates each mixture with SepFormer (Subakan et al., 2021) and transcribes each stream with a SpeechBrain recognizer (Ravanelli et al., 2021), improving SI-SDR by 16.60 dB and reducing WER to 0.40.

Analysis. The gain comes from invoking specialized models rather than from better gluing code. However, the HF-hub agent uses its entire 50-step budget, much of it on integration rather than on the task: reading model cards, installing dependencies, and repairing a failed audio write (Appendix A presents the full trajectories). Model access thus expands what an agent can accomplish, but making that access efficient requires dependable, ready-to-run model interfaces (Section 3.2).

3.2 DEMAND AND CHALLENGES OF BROAD MODEL ACCESS

We conduct an empirical study of Hugging Face to study whether publicly hosted models can be effortlessly used through Transformers `pipeline` API (Wolf et al., 2020). We examine three

aspects. First, to characterize how models are used, we collected 810,572 Transformers models[2] on the hub[3] and examined how usage is distributed across model sizes (O1). Second, to measure how many models can be used through a standard unified interface, we selected *widely-downloaded* Transformers models, ran their officially documented `pipeline` inference code on GPU, and measured the fraction that run without errors (O2). We then analyzed the exceptions raised by the failed models to understand why they fail (O3).

O1: Both small/large models are widely used. Figure 1a illustrates the relationship between model size and usage. First, models across a broad range of sizes, and both small and large models are widely used. Second, a substantial fraction of models receive relatively low usage, with download counts between 10 and 1K.

O2: Many models are unusable. We selected models with downloads exceeding the average (which, after excluding models with zero downloads, stood at 48,755) and sizes smaller than 16 GB. Since `pipeline` supports only Transformers models whose architecture is registered to a task, we evaluate only these nominally supported models. We copied the `pipeline` inference code with dependencies from the official documentations of Hugging Face,[4] to run inference on NVIDIA A40 GPU. Figure 1b shows the fraction of runnable models (i.e., coverage) for each task. Many models failed due to various errors (see below).

O3: No common fix; we need both agents & human experts. We collected those non-runnable models and analyzed the errors. For ease of presentation, we report only error types across all tasks. Figure 1c shows that 1) all tasks have a substantial number of `ImportError` cases, indicating missing task-specific dependencies. 2) Each task still exhibits a diverse set of error types, and different tasks show distinct error distributions. For example, the most frequent error type for image classification models is `ImportError`, whereas for token classification models it is `ValueError`. Importantly, Figure 1c reports failures only at the coarse level of exception types; even within the same category, models exhibit different error messages and root causes, making a uniform category-level repair strategy unlikely to generalize. Addressing these model-specific failures at scale therefore requires agents to construct tailored inference programs, which must then be verified.

3.3 Model inference requires more than binary verification

We use a case study to show that a coding agent can produce an inference program that executes successfully and returns an intended output, yet silently departs from the model's intended inference semantics. We asked Mini-SWE-agent (Yang et al., 2024), to write an inference program for `Supabase/all-MiniLM-L6-v2`[5] on GPU, an ONNX export of a sentence-embedding model (complete generated code is available at Appendix B). The program constructs the workflow itself, including tokenization, ONNX execution, mean pooling, and L2 normalization, and returns a valid 384-dimensional embedding, so output-based verification accepts it. However, although the program requests `CUDAExecutionProvider` for the requested `cuda:0` device, its dependencies install the CPU-only `onnxruntime` package, so ONNX Runtime silently falls back to CPU. This deviation is invisible in the returned embedding, and whether it matters depends on the user's latency requirements. Verification must therefore examine the program, its dependencies, and its execution path, not only its output, and report localized deviations rather than a single pass/fail verdict (Section 4.2).

4 Bolo's Inference Pipeline Construction and Verification

This section describes how Bolo constructs (Section 4.1) and verifies (Section 4.2) inference programs; Figure 2 summarizes the workflow.

4.1 Inference programs generation

Recent coding agents solve repository-level software engineering tasks by following the ReAct paradigm (Yao et al., 2023) in an interactive environment (Yang et al., 2024; Wang et al., 2025).

[2] Indicated by the tags of each model.

[3] Models were collected as of February 6, 2026.

[4] `https://huggingface.co/docs/transformers/en/tasks/<TASK_NAME>`

[5] `https://huggingface.co/Supabase/all-MiniLM-L6-v2`

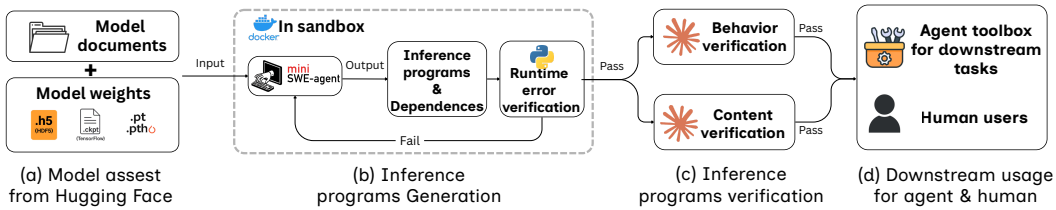


Figure 2: Overview of BOLO. (a) BOLO takes a model's documentation and weight files from Hugging Face as input. (b) Inside a sandbox, a coding agent generates an inference program and its dependencies, iteratively repairing the program until it executes without runtime errors. (c) Each executable program then undergoes behavioral verification, which judges whether its outputs are valid for the model's task, and content verification, which localizes deviations from the model's intended inference semantics. (d) Verified programs are exposed through a unified interface as tools for downstream agents and human users.

These agents are typically developed on benchmarks that provide pre-configured environments and judge success with held-out tests independent of the agent, assumptions that inference-program construction violates. Without such an oracle, an agent can check only whether the program runs, which does not establish faithfulness to the model's intended inference semantics.

BOLO addresses this limitation by decoupling generation from verification. We define generation success solely as runtime executability, a criterion the agent can check reliably. Faithfulness is assessed by an independent verification stage (Section 4.2), so the agent that writes a program never certifies its correctness. To guard against unsafe operations, all agent actions run inside an isolated sandbox, and the safety of each resulting program is further assessed during content verification. Appendix C elaborates on the details of the BOLO's unified inference interface design.

4.2 INFERENCE PROGRAMS VERIFICATION

Verification assesses whether a program faithfully implements the model's *intended inference workflow*, i.e., its documented processing steps; for example, a program that removes a quantized model's quantization backend and loads it in full precision does not follow the intended workflow. Since such a program can still produce plausible outputs, BOLO verifies both (i) what the program produces and (ii) how it produces it. Let p be a program generated for model r, c is execution context, such as the requested *platform* (e.g., hardware device and runtime framework), and $\mathcal{T}_r = \{(x_i, q_i)\}$ a set of *probes*, i.e., test cases that each pair an input x_i with an output check $q_i : \mathcal{Y} \to \{0, 1\}$. We define

$$B(p; r, c) = \mathbf{1}\big[\forall (x_i, q_i) \in \mathcal{T}_r : \ \mathrm{Exec}(p, x_i; c)\!\downarrow \wedge\, q_i\big(\mathrm{Exec}(p, x_i; c)\big) = 1\big]\,, \tag{1}$$

where $\downarrow$ denotes execution without a runtime error. We call p *behaviorally valid* if $B(p; r, c) = 1$. However, a behaviorally valid program may still contain *deviations* from the intended workflow, such as the removed quantization backend., as described in Section 3.3.

Detecting such deviations requires inspecting the program itself, yet even combining static analysis with LLM judges is insufficient. Static analyses, including LLM-assisted auditors (Li et al., 2025; Guo et al., 2025), check formally specified properties or known defect classes (Kildall, 1973; Cousot & Cousot, 1977), while LLM judges collapse their assessment into a single verdict (Wang et al., 2026); neither captures model-specific intent, which is expressed informally in model cards and documentation. For example, the program in Section 3.3 requests the CUDA provider but installs a CPU-only ONNX Runtime: it contains no known defect, yet silently runs on CPU. We therefore design BOLO's content verification as a dimension-conditioned auditor that produces localized evidence against model-specific intent:

$$E_d(p; r, c) = A_\theta(p, \rho_r, c; d) = \{(\ell_j, \delta_j)\}_j, \qquad d \in \mathcal{D}, \tag{2}$$

where ρ_r is the model context, ℓ_j a span of program lines, and δ_j a description of the deviation, with $\mathcal{D} = \{\textsc{Identity}, \textsc{Completeness}, \textsc{Data}, \textsc{Platform}, \textsc{Safety}\}$. These dimensions respectively examine whether the intended artifacts are used, the inference workflow is complete, real inputs flow through that workflow, platform in c is used, and side effects remain within the permitted scope. With

$E = \bigcup_{d \in \mathcal{D}} E_d$, the two signals are complementary:

$$B(p; r, c) = 1 \not\Rightarrow E(p; r, c) = \emptyset, \qquad E(p; r, c) = \emptyset \not\Rightarrow B(p; r, c) = 1. \tag{3}$$

Behavioral validity does not imply fidelity, a clean audit does not imply valid execution, and a reported deviation does not necessarily make the program unfit for use.

Finally, BOLO canonicalizes each evidence item into an action–effect pair, groups similar pairs per dimension, and has an LLM validate each group's generalized pattern, yielding patterns Π. The verification interface thus returns $\mathcal{V}(p; r, c) = \big(B, \{E_d\}_{d \in \mathcal{D}}, \Pi\big)$ rather than a binary verdict.

5 EVALUATION

5.1 MODEL COLLECTION AND EVALUATION SCOPE

All experiments use a fixed snapshot of the Hugging Face models collection described in Section 3.2. Unlike that study, which considers only Transformers models, this snapshot also includes models from non-Transformers libraries (e.g., `llama.cpp`). After excluding models with zero downloads, we retain those with download counts exceeding the average of the remaining models and sizes smaller than 16 GB. Models are divided into two classes according to whether they are nominally supported by the Hugging Face `pipeline` API: CLS-I comprises pipeline-supported models, whereas CLS-II comprises all remaining models, including Transformers models without `pipeline` support and models from other libraries.. This procedure yields 5,639 CLS-I and 2,931 CLS-II models, for a total of 8,570. Appendix D provides the complete selection criteria and exclusion process.

To generate model-specific inference programs, we use Mini-SWE-agent (Yang et al., 2024) as the coding agent, powered by GPT-5.4-mini. For each model, the agent has a 50-step turn limit and a 20-minute time limit. We use an NVIDIA A40 GPU with CUDA 13.0.

5.2 RESEARCH QUESTIONS AND KEY FINDINGS

We evaluate BOLO along three dimensions: its downstream utility for agents, the breadth of its unified inference interface, and the quality of its content verification. Our main findings are:

- **More capable and efficient agents.** Across five downstream tasks, agents equipped with BOLO produce higher-quality task outputs than those given raw Hugging Face Hub access, while reducing total LLM cost by up to 26%.
- **Broader unified model inference access.** Across 8,570 models, BOLO raises executable coverage from 41.6% to 65.2%, including 1,195 of 2,931 models outside `pipeline` support.
- **More informative content verification.** Among 2,019 intended-output programs, BOLO reports no deviation evidence for 88.6%, versus 57.2% receiving a clean binary verdict, while exposing substantive deviations missed by behavioral testing.

5.3 MORE CAPABLE AND EFFICIENT AGENTS WITH BOLO

Detailed tasks. We evaluate agents on five tasks (**W1–5**). (**W1: Denoise and transcribe**) We construct 15 noisy audios that the agent must denoise and transcribe into separate text files. The recordings are selected from the LibriSpeech dataset (Panayotov et al., 2015), each lasting 10–20 seconds, and are corrupted with non-speech noise from the MS-SNSD dataset (Reddy et al., 2019). (**W2: Separate and transcribe**) We construct 10 two-speaker mixtures from audios by 20 distinct LibriSpeech speakers (Panayotov et al., 2015). The agent must separate each mixture into one audio per speaker and transcribe each audio into a text file. (**W3: Detect and narrate**) We sample 25 images from the Microsoft COCO 2017 dataset (Lin et al., 2015), each containing two to six annotated objects. The agent must identify these objects and produce an audio narration describing them. (**W4: Multi-modal retrieval**) We sample 5,000 products with images and textual attributes from the Amazon Berkeley Objects dataset (Collins et al., 2022) and construct 400 queries using alternative images and rephrased attributes of these products. The agent must build separate visual and textual indexes and fuse their retrieval rankings for each image–text query. Finally, **W5: Time-series forecasting**: we sample 12,000 daily sales series from the M5 Forecasting Accuracy dataset (Makridakis et al., 2022a;b). The agent must predict 0.1, 0.5, and 0.9 quantile forecasts over the next 28 days, given 512 days of sales history.

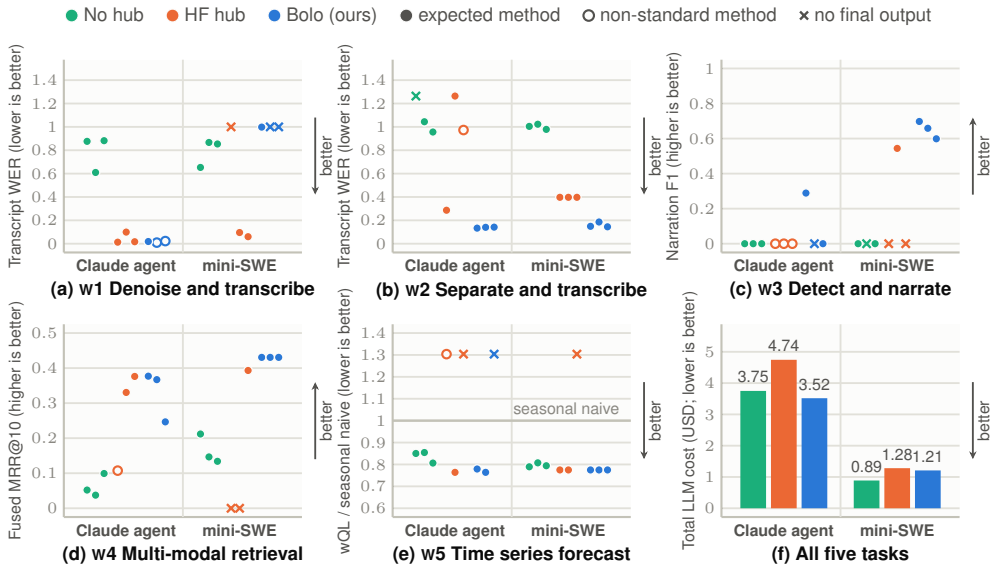


Figure 3: Agent performance (a)–(e) and total LLM cost (f) across five model-dependent workflows. Each point denotes one run; filled and hollow markers indicate expected and non-standard solution methods, respectively, and crosses indicate runs without a final output.

Compared agents and setup. We compare three model-access settings. *No hub* provides no external model access. *HF hub* provides access to 1,578 models from Section 5.1, including their model cards and weights, but requires agents to construct inference code; and BOLO provides the same model collection and search interface, augmented with verified inference programs and dependencies. We evaluate all three settings with two agents: a Claude agent powered by Claude-Haiku-4.5[6] and Mini-SWE-agent (Yang et al., 2024) powered by GPT-5.1-Codex-mini. For each task, all runs share the same task prompt and sandbox, with network access restricted to downloading assets for the selected models. Each *agent-setting-task* configuration is repeated three times.

Metrics. We measure **w1–2** by transcript word error rate (WER; lower is better), **w3** by F1 between narrated and reference objects (higher is better), **w4** by fused MRR@10 (higher is better), and **w5** by weighted quantile loss (wQL) relative to a seven-day seasonal-naive forecast (lower is better; 1.0 denotes baseline performance). We also record total LLM cost and classify each *HF hub* and BOLO run as using the expected method, a non-standard method, or producing no final output. Non-standard methods bypass the provided models, e.g., using TF–IDF instead of embedding models for w4.

Output quality. Figures 3a–3e report output quality across the five tasks. On **w2** (speaker separation), agents with BOLO achieve WERs of 0.133–0.186, compared with 0.286–0.397 for successful *HF Hub* runs using the *expected method* and 0.956–1.044 for *No Hub*. On **w3** (object narration), Mini-SWE-agent with BOLO achieves F1 scores of 0.599–0.698 across all three runs; *No Hub* scores zero, while *HF Hub* reaches 0.544 only through a *non-standard method*. On **w4** (multimodal retrieval), Mini-SWE-agent with BOLO consistently achieves 0.431 MRR@10, whereas only one *HF Hub* run uses the *expected method*, reaching 0.393, and the other two produce no output. *No Hub* achieves only 0.134–0.212. The difference is smaller for Claude agent, whose successful BOLO and *HF Hub* runs overlap. Gains are not universal: on **w1** (denoising), both model-enabled settings can attain low WERs, although BOLO produces no output in two Mini-SWE-agent runs; on **w5** (forecasting), successful model-enabled runs achieve similar normalized wQL values of 0.764–0.779, although BOLO uses the *expected method* in five of six runs, compared with three of six for *HF Hub*. These results suggest that BOLO's advantage comes from more reliable model invocation rather than universally stronger models: its gains are largest when model-specific inference code is critical, but narrow when both settings successfully execute suitable models.

[6]Implemented via `https://github.com/anthropics/claude-agent-sdk-python`.

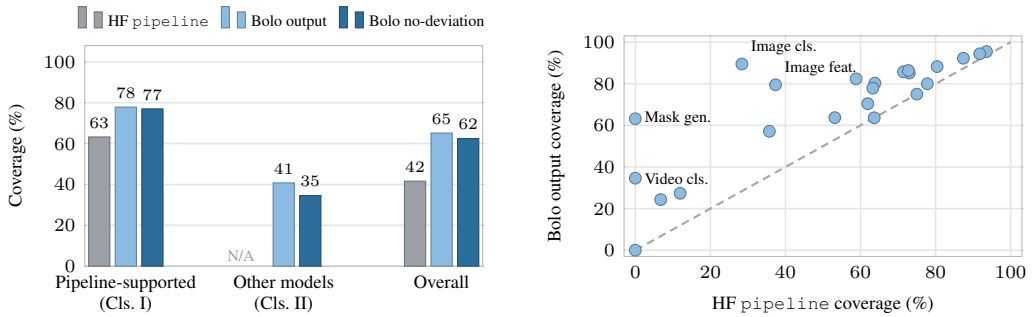
LLM cost. BOLO costs less than *HF Hub* for both agents: $3.52 versus $4.74 with Claude Code and $1.21 versus $1.28 with Mini-SWE-agent. This reduction is consistent with verified inference programs reducing the LLM effort required to construct and repair model-specific inference code.

5.4 BROADER UNIFIED MODEL INFERENCE ACCESS WITH BOLO

Detailed questions. We study two questions: (1) how much does BOLO increase `pipeline` API coverage for CLS-I models; and (2) the coverage BOLO provides for the remaining CLS-II models, to which `pipeline` does not apply.

Compared interfaces and setup. *HF* `pipeline` invokes the standard task-level interface, using the code and dependencies given in the official documentation. BOLO instead exposes its own unified interface, backed differently for the two classes: for CLS-I models, it builds on the `pipeline` API, patching it when it fails; for CLS-II models, it generates a model-specific inference program. For each model, we run every applicable interface in the same environment on the same inputs, which we collected in advance for each task and hold fixed across interfaces.

Metric. For *HF* `pipeline`, coverage is defined as the fraction of evaluated models whose inference program runs without runtime errors. For BOLO, we report two measures. *Output coverage* is the fraction whose program runs to completion and produces an intended output under task-specific behavioral checks. An LLM judge (Claude-Sonnet-5) performs these checks, rejecting task-inconsistent outputs without requiring high downstream accuracy. *No-deviation coverage* additionally requires that content verification report no deviations.



(a) Aggregate coverage.

(b) Coverage across 22 CLS-I tasks.

Figure 4: Model coverage. (a) `pipeline` coverage requires error-free execution. BOLO output coverage requires an intended output. Finally, BOLO's no-deviation coverage also requires no reported deviation evidence. Because `pipeline` does not support CLS-II , the overall baseline counts those models as uncovered. (b) Each point is one pipeline-supported task, and the dashed diagonal denotes equal coverage. BOLO improves coverage on 19 tasks and matches on the remaining three.

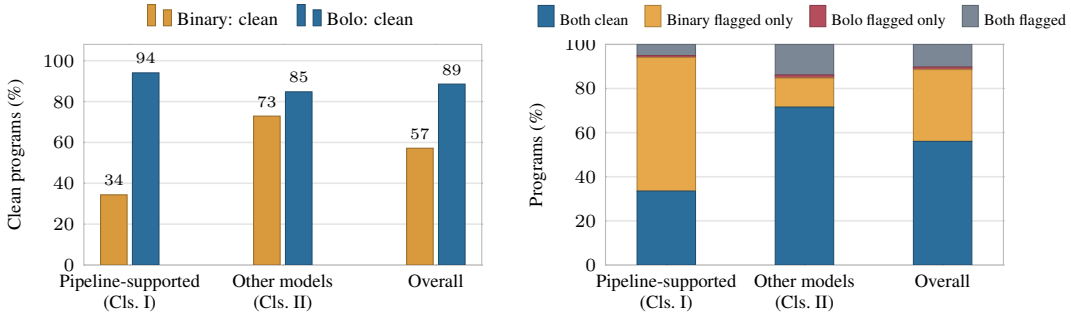
Results. As Figure 4a shows, for CLS-I models, BOLO covers 4,391 of 5,639 models, 824 more than `pipeline` (3,567); only 49 of these have reported deviation evidence. For CLS-II models, where `pipeline` does not apply, BOLO covers 1,195 of 2,931 models, 1,013 of them without deviation evidence; the remaining 182 still execute and come with evidence users can interpret. Overall, BOLO raises coverage from 3,567 to 5,586 of 8,570 models, expanding access within the pipeline-supported population and extending a unified interface to models `pipeline` cannot serve. Per task (Figure 4b), BOLO improves coverage on 19 of 22 CLS-I tasks and matches `pipeline` on the rest, as measured by *Output coverage*.

5.5 MORE INFORMATIVE CONTENT VERIFICATION WITH STRUCTURED EVIDENCE

Detailed questions and setup. We test whether BOLO content verification produces better-supported assessments than a holistic LLM-as-a-judge baseline that labels each implementation correct or incorrect (*Binary*). Comparing only how often the methods raise concerns is insufficient, since fewer concerns may reflect weaker scrutiny rather than better verification. We therefore study two

dimensions: **Q1:** whether BOLO identifies supported deviations missed by *Binary*, and **Q2:** whether BOLO avoids unsupported concerns raised by *Binary*. We compare them on the 2,019 programs passing behavioral checks (824 from CLS-I ; 1,195 from CLS-II). Both methods use Claude-Sonnet-5 with the same program and model context. A program is *flagged* if *Binary* labels it incorrect or BOLO reports any deviation, and *clean* otherwise.

Metrics. In the quantitative study, we report the fraction of programs each method judges clean. In the qualitative study, we manually inspect programs from each disagreement group: for programs only BOLO flags, whether the reported deviations are present in the code (**Q1**); for programs BOLO finds clean but the *Binary* flags, whether the *Binary*'s stated concern holds in the code, i.e., whether BOLO accepted a program with a real deviation (**Q2**).



(a) Clean programs per method.

(b) Joint outcomes.

Figure 5: Content-verification results on 2,019 programs that pass behavioral checks. (a) Fraction of programs each method finds clean. (b) Overall, 1,131 programs (56.0%) are clean under both methods, 657 are flagged only by *Binary*, 23 only by BOLO, and 208 (10.3%) by both.

Quantitative results As Figure 5a shows, the *Binary* finds 57.2% of the 2,019 programs clean, whereas BOLO finds 88.6% clean. Figure 5b shows that the two methods agree on 1,339 programs and there are 657 programs are flagged only by the *Binary*, and only 23 only by BOLO.

Table 1: Qualitative/manual analysis of two disagreement directions on intended-output programs.

Disagreement	Set	Checked	Case-analysis result
Programs that only BOLO flags	23	10	Deviation present in 7/10; changes behavior in 5/10.
Programs that only *Binary* flags	657	10	*Binary*'s stated concern holds in 0/10; 1/10 has a deviation neither method identifies.

Qualitative results. For **Q1**, most BOLO-only flags are genuine (Table 1), and several change what the program computes despite passing every output check: the `Xenova/detr-resnet-50` program downloads the ONNX artifact but runs `facebook/detr-resnet-50` in PyTorch, the `WENGSYX/Multilingual_SimCSE` program omits the documented mean pooling. Two others substitute an upstream image processor with no demonstrated effect, which BOLO reports for users to judge; the three unsupported flags concern harmless or hypothetical issues. For **Q2**, none of the *Binary*'s concerns holds on the sampled programs; e.g., it claims the `abhijit2111/Pic2Story` program loads the wrong repository, though it loads the repository's own weights. One deviation escapes both methods: the `yandex/YandexGPT-5-Lite-8B-instruct-GGUF` program requests GPU but installs a CPU-only `llama.cpp`. Detailed programs are in Appendix E.

6 LIMITATIONS AND FUTURE DIRECTIONS

Consistent with our definition of verification as evidence rather than proof, BOLO's content verification may occasionally miss or over-report deviations (Section 5.5), and our evaluation covers widely-downloaded Hugging Face models under 16 GB on a single GPU type. In future work, BOLO's inference programs collection via a shared interface with enumerable parameters (Appendix D) makes it a natural platform for tuning each model's inference parameters for best performance, which is otherwise difficult across heterogeneous models.

AI USE STATEMENT

In this work, we used generative AI tools for: help develop theoretical models or conceptual frameworks, formulate mathematical claims, design or provide feedback on research methodology or experiments, implement methods, assist with translation, support qualitative and thematic data analysis, and interpret results. We have not used generative AI tools for: generate synthetic data sets, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, clean and reformat dataset are not applicable to this work. Additionally, we used generative AI tools for create or modify scientific figures or images. We have reviewed all AI-assisted work. For example, we ask coding agent to implement the BOLO generation and verification methods, we then review the code by manual inspection as well as design unit tests to verify the generated code. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. Gradio: Hassle-free sharing and testing of ml models in the wild, 2019. URL https://arxiv.org/abs/1906.02569.

Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models, 2022. URL https://arxiv.org/abs/2108.07258.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022. URL https://arxiv.org/abs/2207.10397.

Jasmine Collins, Shubham Goel, Kenan Deng, Achleshwar Luthra, Leon Xu, Erhan Gundogdu, Xi Zhang, Tomas F. Yago Vicente, Thomas Dideriksen, Himanshu Arora, Matthieu Guillaumin, and Jitendra Malik. Abo: Dataset and benchmarks for real-world 3d object understanding, 2022. URL https://arxiv.org/abs/2110.06199.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pp. 238–252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973. URL https://doi.org/10.1145/512950.512973.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous llm-agent for repository-level code auditing, 2025. URL https://arxiv.org/abs/2501.18160.

D. Huggins-Daines, M. Kumar, A. Chan, A.W. Black, M. Ravishankar, and A.I. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, volume 1, pp. I–I, 2006. doi: 10.1109/ICASSP.2006.1659988.

Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pp. 194–206, New York, NY, USA, 1973. Association for Computing Machinery. ISBN 9781450373494. doi: 10.1145/512927.512945. URL https://doi.org/10.1145/512927.512945.

Chenliang Li, Hehong Chen, Ming Yan, Weizhou Shen, Haiyang Xu, Zhikai Wu, Zhicheng Zhang, Wenmeng Zhou, Yingda Chen, Chen Cheng, Hongzhu Shi, Ji Zhang, Fei Huang, and Jingren Zhou. Modelscope-agent: Building your customizable agent system with open-source large language models, 2023. URL https://arxiv.org/abs/2309.00986.

Ziyang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities, 2025. URL https://arxiv.org/abs/2405.17238.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, Yun Wang, Linjun Shou, Ming Gong, and Nan Duan. Taskmatrix.ai: Completing tasks by connecting foundation models with millions of apis, 2023. URL https://arxiv.org/abs/2303.16434.

Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015. URL https://arxiv.org/abs/1405.0312.

Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. Fvel: Interactive formal verification environment with large language models via theorem proving, 2024. URL https://arxiv.org/abs/2406.14408.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572. Curran Associates, Inc., 2023. doi: 10.52202/075280-0943. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models, 2023. URL https://arxiv.org/abs/2304.09842.

Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. M5 accuracy competition: Results, findings, and conclusions. *International Journal of Forecasting*, 38(4):1346–1364, 2022a. ISSN 0169-2070. doi: https://doi.org/10.1016/j.ijforecast.2021.11.013. URL https://www.sciencedirect.com/science/article/pii/S0169207021001874. Special Issue: M5 competition.

Spyros Makridakis, Evangelos Spiliotis, Vassilios Assimakopoulos, Zhi Chen, Anil Gaba, Ilia Tsetlin, and Robert L. Winkler. The m5 uncertainty competition: Results, findings and conclusions. *International Journal of Forecasting*, 38(4):1365–1385, 2022b. ISSN 0169-2070. doi: https://doi.org/10.1016/j.ijforecast.2021.10.009. URL https://www.sciencedirect.com/science/article/pii/S0169207021001722. Special Issue: M5 competition.

Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5206–5210, 2015. doi: 10.1109/ICASSP.2015.7178964.

Mirco Ravanelli, Titouan Parcollet, Peter Plantinga, Aku Rouhe, Samuele Cornell, Loren Lugosch, Cem Subakan, Nauman Dawalatabad, Abdelwahab Heba, Jianyuan Zhong, Ju-Chieh Chou, Sung-Lin Yeh, Szu-Wei Fu, Chien-Feng Liao, Elena Rastorgueva, François Grondin, William Aris,

Hwidong Na, Yan Gao, Renato De Mori, and Yoshua Bengio. Speechbrain: A general-purpose speech toolkit, 2021. URL https://arxiv.org/abs/2106.04624.

Chandan KA Reddy, Ebrahim Beyrami, Jamie Pool, Ross Cutler, Sriram Srinivasan, and Johannes Gehrke. A scalable noisy speech dataset and online subjective test framework. *Proc. Interspeech 2019*, pp. 1816–1820, 2019.

Jonathan Le Roux, Scott Wisdom, Hakan Erdogan, and John R. Hershey. Sdr - half-baked or well done?, 2018. URL https://arxiv.org/abs/1811.02508.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face, 2023. URL https://arxiv.org/abs/2303.17580.

Cem Subakan, Mirco Ravanelli, Samuele Cornell, Mirko Bronzi, and Jianyuan Zhong. Attention is all you need in speech separation, 2021. URL https://arxiv.org/abs/2010.13154.

The ModelScope Team. Modelscope: bring the notion of model-as-a-service to life. https://github.com/modelscope/modelscope, 2023.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Yutong Wang, Pengliang Ji, Chaoqun Yang, Kaixin Li, Ming Hu, Jiaoyang Li, and Guillaume Sartoretti. Mcts-judge: Test-time scaling in llm-as-a-judge for code correctness evaluation, 2026. URL https://arxiv.org/abs/2502.12468.

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. Transformers: State-of-the-art natural language processing. In Qun Liu and David Schlangen (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6/.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models, 2023. URL https://arxiv.org/abs/2303.04671.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

A Full Trajectories for Speech Separation Case Study

Section 3.1 compares two Mini-SWE-agent runs on the same task: separate each of ten two-speaker mixtures into two 8-kHz PCM streams and transcribe every stream. The traces below retain every counted agent step while abbreviating commands and outputs.

Hugging-Face-powered agent (HF arm). The agent completed all requested outputs but exhausted its 50-step budget before submitting a final response. Evaluation measured 16.60 dB SI-SDRi and 0.40 WER.

Inference-program deduction overhead. The agent does not begin writing the end-to-end task program until step 46. It spends 30 steps (3–24 and 38–45) locating artifacts, reading model cards, inspecting APIs, and testing how to invoke the two selected SpeechBrain models. Another 12 steps (26–37) are consumed by a discarded Whisper/ASR search branch. Thus, 42 of 50 interactions go to model-specific inference deduction or an unsuccessful candidate, rather than constructing and running the task pipeline.

```
01  Inspect workspace -> find inputs plus search, fetch, and install tools.
02  Search "speech separation" -> rank speechbrain/sepformer-wsj02mix first.

--- SELECTED-MODEL INFERENCE DEDUCTION: SEPFORMER (STEPS 03-24) ---
03  Fetch SepFormer -> cache 10 files (0.11 GB), including checkpoints and README.
04  Try cd fetch/speechbrain/sepformer-wsj02mix -> fail; fetch is not a directory.
05  Inspect fetch -> only the fetch executable is shown.
06  Repeat inspection of fetch -> same result.
07  List hidden workspace entries -> discover .hub request spool.
08  Check working directory -> /workspace.
09  Inspect .hub -> empty after the completed request.
10  Enumerate workspace directories -> only .hub and inputs/mixtures.
11  Search workspace for brain.ckpt -> no result.
12  Enumerate workspace tree -> model files are not stored under /workspace.
13  Identify fetch -> Python executable, not a fetched-model directory.
14  Probe /workspace/.cache -> directory does not exist.
15  Probe /root -> no model cache is shown.
16  Read fetch executable -> learn that requests are relayed through .hub.
17  Inspect filesystem root -> notice the shared /hf directory.
18  Search filesystem for SepFormer -> locate its cache under /hf/hub.
19  Read README at repository-cache root -> fail; README is inside a snapshot.
20  Inspect repository cache -> find blobs, refs, and snapshots.
21  Inspect snapshots -> obtain snapshot hash 3a2826...3531.
22  List snapshot -> confirm README and all SepFormer checkpoint/config files.
23  Install speechbrain -> installation succeeds with its dependencies.
24  Read SepFormer model card -> recover the documented loading/inference pattern.
25  Check HF_HOME -> /hf, confirming the cache location.

--- DISCARDED ASR/WHISPER SEARCH DETOUR (STEPS 26-37) ---
26  Fetch openai/whisper-small.en -> fail because that repository is unavailable.
27  Search "whisper-small" -> return alternatives, but not the requested OpenAI repo.
28  Search "openai/whisper" -> rank Xenova/whisper-tiny.en first.
29  Search "speechbrain asr" -> results are mostly enhancement/other ASR models.
30  Search "wav2vec2" -> results do not expose a suitable standard ASR checkpoint.
31  Search exact OpenAI Whisper name -> still return Xenova/whisper-tiny.en.
32  Search "deepspeech" -> return unrelated or heavier ASR alternatives.
33  Search exact wav2vec2-base-960h name -> return an unrelated model.
34  Search exact Xenova Whisper name -> confirm Xenova/whisper-tiny.en.
35  Fetch Xenova Whisper -> cache 45 files (1.87 GB), mainly ONNX variants.
36  Inspect Xenova snapshots -> obtain snapshot hash 79fb38...049.
37  List Xenova snapshot -> confirm tokenizer/config files and ONNX directory.

--- SELECTED-MODEL INFERENCE DEDUCTION: SPEECHBRAIN ASR (STEPS 38-45) ---
38  Search "speechbrain/librispeech" -> find asr-crdnn-rnnlm-librispeech (rank 5).
39  Fetch SpeechBrain ASR -> cache 9 files (0.69 GB), including ASR and LM weights.
40  Read ASR snapshot with an invalid glob -> fail to resolve a snapshot path.
41  Inspect ASR snapshots -> obtain snapshot hash 979a53...8a3.
42  Read ASR model card -> recover the documented EncoderDecoderASR workflow.
43  Inspect transcribe_file documentation -> confirm file-path input and text output.
44  Test SepFormer on mix_001 -> GPU inference works; output is [1,80160,2] at 8 kHz.
45  Retest SepFormer on CPU -> same source shape; confirms a CPU fallback path.

--- TASK-LEVEL PIPELINE CONSTRUCTION AND EXECUTION (STEPS 46-50) ---
46  Write full pipeline -> SepFormer separation, audio save, then SpeechBrain ASR.
47  Run pipeline -> fail on mix_001 because torchaudio.save requires TorchCodec.
48  Install torchcodec -> installation succeeds.
49  Rerun pipeline -> process all ten mixtures and both sources successfully.
50  List workspace -> verify 20 WAV files and 20 transcript files; budget exhausted.
```

Listing 1: Abbreviated full trajectory of the Hugging-Face-powered agent.

Non-ARM agent (no model-hub access). The agent built a classical signal-processing pipeline and submitted successfully at step 41. Evaluation measured -2.22 dB SI-SDRi and 1.00 WER.

```
01  Inspect input root -> find the mixtures directory.
02  List mixtures -> confirm mix_001.wav through mix_010.wav.
03  Inspect filesystem root -> review the available local environment.
04  Install python_speech_features and scikit-learn -> installation succeeds.
05  Enumerate inputs -> confirm the ten mixture files and no extra assets.
06  Inspect /hf -> no model artifacts are available to this arm.
07  Enumerate workspace directories -> find only inputs and local package paths.
08  Install PocketSphinx -> offline ASR and its dependencies install successfully.
09  Locate PocketSphinx package -> confirm import from /workspace/.pip.
10  Query PocketSphinx model path -> locate the bundled acoustic/language model.
11  Inspect model root -> find the en-us model directory.
12  Inspect en-us assets -> confirm acoustic model, language model, and dictionary.
13  Prototype two-component NMF on mix_001 -> factorization shapes are consistent.
14  Reconstruct two sources -> fail because scipy.istft rejects padded=....
15  Remove unsupported argument and retry -> write two sources for mix_001.
16  Prototype PocketSphinx decoding -> fail by joining AudioFile objects as strings.
17  Inspect decoder iteration with wrong temp path -> fail with FileNotFoundError.
18  List workspace -> confirm two separated WAV files and identify the path mismatch.
19  Write the correctly named 16-kHz temporary WAV -> succeeds.
20  Inspect one decoder item -> obtain a noisy hypothesis and AudioFile object type.
21  Call phrase.hypothesis() explicitly -> confirm the correct transcript extraction.
22  Write end-to-end NMF + PocketSphinx pipeline for all mixtures.
23  Attempt an apply_patch edit -> fail because apply_patch is unavailable in sandbox.
24  Rewrite the pipeline directly with the intended resampling fix.
25  Run all mixtures -> emit 20 WAV/text pairs, with repeated STFT NOLA warnings.
26  List workspace -> confirm outputs and notice a stale temporary WAV.
27  Remove stale temporary WAV -> cleanup succeeds.
28  Inspect mix_001 source-1 transcript -> text is fluent-looking but badly garbled.
29  Rewrite pipeline -> add a transcript writer that always terminates files cleanly.
30  Rerun all mixtures -> succeeds but retains the STFT NOLA warnings.
31  Rewrite pipeline -> use compatible default STFT/ISTFT boundary handling.
32  Rerun all mixtures -> succeeds without the earlier reconstruction warnings.
33  Inspect mix_005 source-1 transcript -> output remains badly garbled.
34  Count generated WAV files -> 20.
35  Inspect WAV metadata -> all are mono, int16, but currently 16 kHz.
36  Inspect an input mixture -> input rate is 16 kHz.
37  Rewrite pipeline -> downsample deliverables to required 8 kHz; upsample for ASR.
38  Rerun all mixtures -> produce all 20 WAV/text pairs successfully.
39  Verify output rates -> all 20 delivered WAV files are 8 kHz.
40  List workspace -> confirm the complete set of 20 WAV and 20 text deliverables.
41  Emit completion signal -> task ends successfully.
```

Listing 2: Abbreviated full trajectory of the non-ARM agent.

B Complete Inference Code for Verification Case Study

Section 3.3 studies the generated program for `Supabase/all-MiniLM-L6-v2`. Listing 3 reproduces the complete generated `inference.py` without omissions. The program requests `CUDAExecutionProvider` for `cuda:0`, but its generated dependency file installs `onnxruntime==1.29.0` (the CPU package) rather than `onnxruntime-gpu`. ONNX Runtime therefore falls back to the CPU provider while the program still returns a valid embedding.

```python
repo_id = "Supabase/all-MiniLM-L6-v2"
device = "cuda:0"
text_1 = "A quick brown fox jumps over the lazy dog."

from pathlib import Path
import numpy as np
from huggingface_hub import hf_hub_download
from transformers import BertTokenizer
import torch
import onnxruntime as ort

onnx_path = hf_hub_download(repo_id=repo_id, filename="onnx/model.onnx")

vocab_path = hf_hub_download(repo_id=repo_id, filename="vocab.txt")

tokenizer = BertTokenizer(vocab_file=vocab_path)
providers = ["CPUExecutionProvider"] if device.startswith("cpu") else ["
    CUDAExecutionProvider", "CPUExecutionProvider"]
session = ort.InferenceSession(onnx_path, providers=providers)
```

```
texts = [text_1] if isinstance(text_1, str) else list(text_1)
inputs = tokenizer(texts, padding=True, truncation=True, return_tensors
    ="np")
ort_inputs = {k: v.astype(np.int64) for k, v in inputs.items()}

outputs = session.run(None, ort_inputs)
last_hidden_state = torch.from_numpy(outputs[0])
attention_mask = torch.from_numpy(inputs["attention_mask"]).unsqueeze
    (-1).type_as(last_hidden_state)
summed = (last_hidden_state * attention_mask).sum(dim=1)
counts = attention_mask.sum(dim=1).clamp(min=1e-9)
embeddings = torch.nn.functional.normalize(summed / counts, p=2, dim=1)
embedding = embeddings[0].detach().cpu().numpy().tolist()
RESULT = {"embedding": embedding}
```

Listing 3: Complete generated inference program for `Supabase/all-MiniLM-L6-v2`.

C Unified Inference Interface Design

BOLO separates a uniform invocation layer from model-specific inference logic. Each supported model is represented by a parameterized Jinja2 inference template together with a model-specific `requirements.txt`. Human users and agents invoke any supported model through the same entry point, `bolo.pipeline(repo_id, **kwargs)`, and can inspect its accepted parameters and default values through `bolo.list_params`. At runtime, BOLO retrieves the corresponding artifacts, renders the template with the supplied inputs and device configuration, executes the resulting program in the model's isolated environment, and returns the object assigned to a common `RESULT` variable. This design standardizes how models are invoked without requiring their heterogeneous loading, preprocessing, inference, and postprocessing procedures to conform to a fixed set of task- or architecture-specific implementations.

D Detailed Model Selection

We collected models on Hugging Face by its tags, if "Transformers" is presented in the tag, the model is considered as Transformers models. There are total 2,250,578 effective models on Hugging Face as of Feb. 6 2026. Then, we further filter models by their functionalities or names. For example, we don't consider the reinforcement learning, robotics models, and PEFT (Parameter-Efficient Fine-Tuning) models. After extensive filtering, we collect 810,572 Transformers models and 182,605 Non-Transformers models. Figure 6 and Figure 7 show the corresponding downloads statistics. For Transformers models, there are 123,370 models with zero downloads[7], and there are 67,405 models in Non-Transformers models with zero downloads. After excluding them, we calculate the average downloads for Transformers models as 48,755 and 25,433 for Non-Transformers models. Then we can only use the models exceeding corresponding average downloads. Note that CLS-I models are only part of Transformers models supported by `pipeline` API and the CLS-II represents remaining models including Transformers and Non-Transformers models.

E Inference Programs Used in Qualitative Study

Section 5.5 manually examines 20 behaviorally validated inference programs sampled from the two directions in which the binary judge and BOLO disagree. This appendix presents the complete `inference.py` used for every inspected program. Within each disagreement direction, the listings follow the study's sample order: five CLS-I programs followed by five CLS-II programs. Generated dependency manifests and execution logs are separate artifacts and are not repeated here. Absolute input paths are retained from the evaluation environment. For compatibility with the current ICLR pdfLaTeX build, the two non-ASCII input strings use equivalent Python Unicode escape notation; this changes their source spelling but not their values.

[7]Their downloads are zero means that Hugging Face cannot trace their downloads.

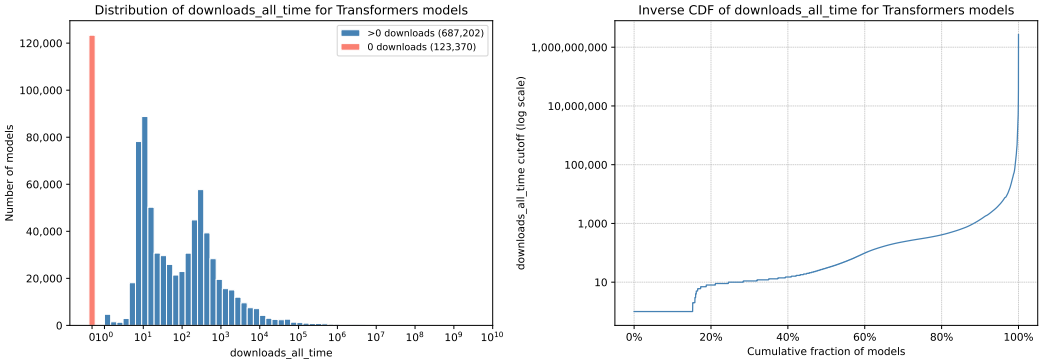


Figure 6: Downloads statistics for collected Transformers models as of Feb. 6 2026.

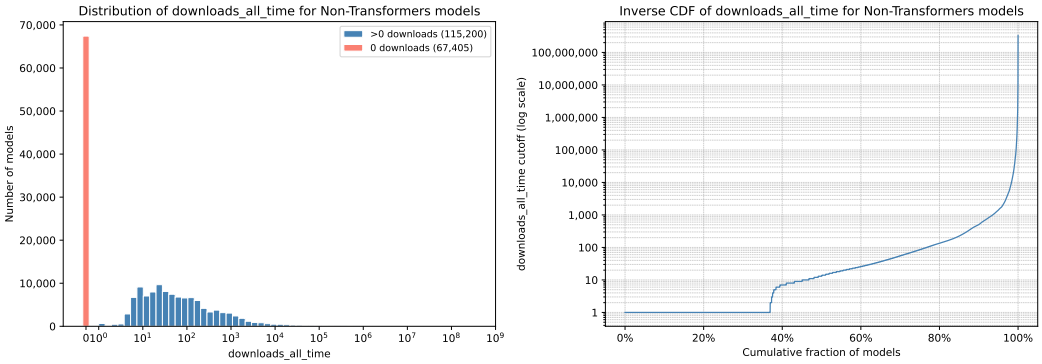


Figure 7: Downloads statistics for collected Non-Transformers models as of Feb. 6 2026.

E.1 Programs Flagged Only by the Binary Judge

The following ten programs are sampled from the 657 intended-output programs that the binary judge flags but BOLO finds clean.

```
task = "text-generation"
model = "unsloth/Meta-Llama-3.1-8B-Instruct-unsloth-bnb-4bit"
device = "cuda:0"
input_prompt = "Hello, my name is Eric, how are you today?"
generation_kwargs = {"max_new_tokens": 256, "temperature": 0.7}

from transformers import pipeline

pipe = pipeline(task=task, model=model, device=device)
raw_outputs = pipe(input_prompt, **generation_kwargs)

RESULT = {"outputs": raw_outputs}
```

Listing 4: Complete CLS-I program for `unsloth/Meta-Llama-3.1-8B-Instruct-unsloth-bnb-4bit`.

```
task = "image-classification"
repo_id = "timm/swin_small_patch4_window7_224.ms_in22k_ft_in1k"
device = "cuda:0"
model = "timm/swin_small_patch4_window7_224.ms_in22k_ft_in1k"
input_image = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"


from PIL import Image
```

```
from transformers import pipeline

pipe = pipeline(task=task, model=model, device=device)

image = Image.open(input_image).convert("RGB")

predictions = pipe(image)

RESULT = {"predictions": predictions}
```

Listing 5: Complete CLS-I program for `timm/swin_small_patch4_window7_224.ms_in22k_ft_in1k`.

```
repo_id = "timm/res2next50.in1k"
device = "cuda:0"
task = "image-classification"
input_data = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"

from transformers import pipeline

pipe = pipeline(task=task, model=repo_id, device=device)
RESULT = pipe(input_data)
```

Listing 6: Complete CLS-I program for `timm/res2next50.in1k`.

```
task = "image-classification"
model = "timm/seresnext26d_32x4d.bt_in1k"
repo_id = "timm/seresnext26d_32x4d.bt_in1k"
device = "cuda:0"
input_image = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"


from transformers import pipeline

pipe = pipeline(task=task, model=model, device=device)

prepared_input = input_image

RESULT = pipe(prepared_input)
```

Listing 7: Complete CLS-I program for `timm/seresnext26d_32x4d.bt_in1k`.

```
task = "image-classification"
model = "timm/convnext_small.fb_in22k"
repo_id = "timm/convnext_small.fb_in22k"
device = "cuda:0"
input_image = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"


from transformers import pipeline

pipe = pipeline(task=task, model=model, device=device)
RESULT = pipe(input_image)
```

Listing 8: Complete CLS-I program for `timm/convnext_small.fb_in22k`.

```
repo_id = "malmarjeh/mbert2mbert-arabic-text-summarization"
device = "cuda:0"
text = "\u0634\u0647\u062f\u062a \u0645\u062f\u064a\u0646\u0629 \u0637\
    u0631\u0627\u0628\u0644\u0633\u060c \u0645\u0633\u0627\u0621 \u0623\
    u0645\u0633 \u0627\u0644\u0623\u0631\u0628\u0639\u0627\u0621\u060c \
    u0627\u062d\u062a\u062c\u0627\u062c\u0627\u062a \u0634\u0639\u0628\
    u064a\u0629 \u0648\u0623\u0639\u0645\u0627\u0644 \u0634\u063a\u0628
```

```
    \u0644\u0644\u064a\u0648\u0645 \u0627\u0644\u062b\u0627\u0644\u062b
    \u0639\u0644\u0649 \u0627\u0644\u062a\u0648\u0627\u0644\u064a\u060c
    \u0648\u0630\u0644\u0643 \u0628\u0633\u0628\u0628 \u062a\u0631\u062f
    \u064a \u0627\u0644\u0648\u0636\u0639 \u0627\u0644\u0645\u0639\u064a
    \u0634\u064a \u0648\u0627\u0644\u0627\u0642\u062a\u0635\u0627\u062f\
    u064a. \u0648\u0627\u0646\u062f\u0644\u0639\u062a \u0645\u0648\u0627
    \u062c\u0647\u0627\u062a \u0639\u0646\u064a\u0641\u0629 \u0648\u0639
    \u0645\u0644\u064a\u0627\u062a \u0643\u0631 \u0648\u0641\u0631 \
    u0645\u0627 \u0628\u064a\u0646 \u0627\u0644\u062c\u064a\u0634 \u0627
    \u0644\u0644\u0628\u0646\u0627\u0646\u064a \u0648\u0627\u0644\u0645\
    u062d\u062a\u062c\u064a\u0646 \u0627\u0633\u062a\u0631\u062a \
    u0644\u0633\u0627\u0639\u0627\u062a\u060c \u0625\u062b\u0631 \u0645\
    u062d\u0627\u0648\u0644\u0629 \u0641\u062a\u062d \u0627\u0644\u0637\
    u0631\u0642\u0627\u062a \u0627\u0644\u0645\u0642\u0637\u0648\u0639\
    u0629\u060c \u0645\u0627 \u0623\u062f\u0649 \u0625\u0644\u0649 \
    u0625\u0635\u0627\u0628\u0629 \u0627\u0644\u0639\u0634\u0631\u0627\
    u062a \u0645\u0646 \u0627\u0644\u0637\u0631\u0641\u064a\u0646."
max_length = 200
num_beams = 3
repetition_penalty = 3.0
length_penalty = 1.0
no_repeat_ngram_size = 3

import torch
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from arabert.preprocess import ArabertPreprocessor

model = AutoModelForSeq2SeqLM.from_pretrained(repo_id).to(device)
tokenizer = AutoTokenizer.from_pretrained(repo_id, use_fast=False,
    fix_mistral_regex=True)
preprocessor = ArabertPreprocessor(model_name="")
text = preprocessor.preprocess(text)
inputs = tokenizer(text, return_tensors="pt", truncation=True).to(
    device)
output_ids = model.generate(**inputs, num_beams=num_beams,
    repetition_penalty=repetition_penalty, max_length=max_length,
    length_penalty=length_penalty, no_repeat_ngram_size=
    no_repeat_ngram_size, pad_token_id=tokenizer.eos_token_id)
output = tokenizer.decode(output_ids[0], skip_special_tokens=True)

RESULT = {"output": output}
```

Listing 9: Complete CLS-II program for `malmarjeh/mbert2mbert-arabic-text-summarization`.

```
repo_id = "yandex/YandexGPT-5-Lite-8B-instruct-GGUF"
device = "cuda:0"
quant_filename = "YandexGPT-5-Lite-8B-instruct-Q4_K_M.gguf"
prompt = "\u041f\u0440\u0438\u0432\u0435\u0442! \u041a\u0440\u0430\
    u0442\u043a\u043e \u043e\u0431\u044a\u044f\u0441\u043d\u0438, \u0447
    \u0442\u043e \u0442\u0430\u043a\u043e\u0435 \u043c\u0430\u0448\u0438
    \u043d\u043d\u043e\u0435 \u043e\u0431\u0443\u0447\u0435\u043d\u0438\
    u0435."

import os
from huggingface_hub import hf_hub_download
from llama_cpp import Llama

model_path = hf_hub_download(repo_id=repo_id, filename=quant_filename)

n_gpu_layers = -1 if str(device).startswith("cuda") else 0
llm = Llama(
    model_path=model_path,
    n_gpu_layers=n_gpu_layers,
```

```
    n_ctx=2048,
    verbose=False,
)

messages = [
    {"role": "user", "content": prompt},
]

response = llm.create_chat_completion(
    messages=messages,
    temperature=0.2,
    top_p=0.9,
    max_tokens=256,
)

output_text = response["choices"][0]["message"]["content"]

RESULT = {"output_text": output_text}
```

Listing 10: Complete CLS-II program for `yandex/YandexGPT-5-Lite-8B-instruct-GGUF`.

```
repo_id = "RedHatAI/phi-4-quantized.w4a16"
device = "cuda:0"
prompt = "Write a short introduction to large language models."

import os
import torch
from huggingface_hub import hf_hub_download
from transformers import AutoTokenizer, AutoModelForCausalLM

config_path = hf_hub_download(repo_id=repo_id, filename="config.json")
model_index_path = hf_hub_download(repo_id=repo_id, filename="model.
    safetensors.index.json")
model_1_path = hf_hub_download(repo_id=repo_id, filename="model-00001-
    of-00002.safetensors")
model_2_path = hf_hub_download(repo_id=repo_id, filename="model-00002-
    of-00002.safetensors")

tokenizer = AutoTokenizer.from_pretrained(repo_id)
model = AutoModelForCausalLM.from_pretrained(
    repo_id,
    torch_dtype=torch.bfloat16,
    device_map=None,
)
model.to(device)
model.eval()

messages = [
    {"role": "user", "content": prompt},
]
chat_text = tokenizer.apply_chat_template(messages,
    add_generation_prompt=True, tokenize=False)
inputs = tokenizer(chat_text, return_tensors="pt")
input_ids = inputs["input_ids"].to(device)

with torch.no_grad():
    generated = model.generate(
        input_ids=input_ids,
        max_new_tokens=128,
        do_sample=False,
        pad_token_id=tokenizer.eos_token_id,
    )
```

```
output_text = tokenizer.decode(generated[0][input_ids.shape[-1]:],
    skip_special_tokens=True)

RESULT = {"text": output_text}
```

Listing 11: Complete CLS-II program for `RedHatAI/phi-4-quantized.w4a16`.

```
repo_id = "abhijit2111/Pic2Story"
device = "cuda:0"
text_prompt = "a photography of"
image_path = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"

import os
import requests
import torch
from PIL import Image
from transformers import BlipProcessor, BlipForConditionalGeneration
from huggingface_hub import hf_hub_download

config_path = hf_hub_download(repo_id=repo_id, filename="config.json")
processor_config_path = hf_hub_download(repo_id=repo_id, filename="
    preprocessor_config.json")
tokenizer_config_path = hf_hub_download(repo_id=repo_id, filename="
    tokenizer_config.json")
special_tokens_map_path = hf_hub_download(repo_id=repo_id, filename="
    special_tokens_map.json")
vocab_path = hf_hub_download(repo_id=repo_id, filename="vocab.txt")
tokenizer_json_path = hf_hub_download(repo_id=repo_id, filename="
    tokenizer.json")
weights_path = hf_hub_download(repo_id=repo_id, filename="pytorch_model.
    bin")

processor = BlipProcessor.from_pretrained(os.path.dirname(
    processor_config_path))
model = BlipForConditionalGeneration.from_pretrained(os.path.dirname(
    weights_path), local_files_only=True)
model = model.to(device)
model.eval()

if image_path.startswith(("http://", "https://")):
    raw_image = Image.open(requests.get(image_path, stream=True).raw).
    convert("RGB")
else:
    raw_image = Image.open(image_path).convert("RGB")

inputs = processor(raw_image, text_prompt, return_tensors="pt")
inputs = {k: v.to(device) for k, v in inputs.items()}

with torch.no_grad():
    out = model.generate(**inputs, max_new_tokens=30)

caption = processor.decode(out[0], skip_special_tokens=True)

RESULT = {"caption": caption}
```

Listing 12: Complete CLS-II program for `abhijit2111/Pic2Story`.

```
repo_id = "fal/AuraSR-v2"
device = "cuda:0"
image_path = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"

import json
from pathlib import Path
from io import BytesIO
```

```python
from PIL import Image
import requests
import torch
from huggingface_hub import hf_hub_download
from aura_sr import AuraSR

model_path = hf_hub_download(repo_id=repo_id, filename="model.
    safetensors")
config_path = hf_hub_download(repo_id=repo_id, filename="config.json")

if image_path.startswith("http://") or image_path.startswith("https://")
    :
    response = requests.get(image_path)
    image = Image.open(BytesIO(response.content)).convert("RGB")
else:
    image = Image.open(image_path).convert("RGB")

image = image.resize((256, 256))

model = AuraSR.from_pretrained(repo_id)
upscaled_image = model.upscale_4x_overlapped(image)

RESULT = {"upscaled_image": upscaled_image}
```

Listing 13: Complete CLS-II program for `fal/AuraSR-v2`.

E.2 PROGRAMS FLAGGED ONLY BY BOLO

The following ten programs are sampled from the 23 intended-output programs that BOLO flags but the binary judge finds clean.

```python
task = "text-generation"
input_prompt = "Hello, my name is Eric, how are you?"
tokenizer_repo = "bigscience/bloom-7b1"
repo_id = "bigscience/bloom-7b1-intermediate"
device = "cuda:0"

from transformers import pipeline

pipe = pipeline(
    task=task,
    model=repo_id,
    tokenizer=tokenizer_repo,
    device=device,
)
RESULT = pipe(input_prompt)
```

Listing 14: Complete CLS-I program for `bigscience/bloom-7b1-intermediate`.

```python
repo_id = "Veggiee/roberta-base-gptq-4bit"
device = "cuda:0"
task = "fill-mask"
masked_prompt = "Paris is the {mask} of France."

import os
from transformers import pipeline

pipeline_kwargs = {"task": task, "model": repo_id}
if device == "cpu":
    pipeline_kwargs["device"] = "cpu"
elif device.startswith("cuda"):
    cuda_index = device.split(":", 1)[1] if ":" in device else "0"
    os.environ["CUDA_VISIBLE_DEVICES"] = cuda_index
else:
```

```
    pipeline_kwargs["device"] = device

pipe = pipeline(**pipeline_kwargs)
mask_token = pipe.tokenizer.mask_token
text = masked_prompt.replace("{mask}", mask_token)
RESULT = pipe(text)
```

Listing 15: Complete CLS-I program for `Veggiee/roberta-base-gptq-4bit`.

```
task = "feature-extraction"
repo_id = "WENGSYX/Multilingual_SimCSE"
device = "cuda:0"
input_prompt = "This is a test sentence for SimCSE feature extraction."

from transformers import pipeline

pipe = pipeline(task=task, model=repo_id, device=device)
RESULT = pipe(input_prompt)
```

Listing 16: Complete CLS-I program for `WENGSYX/Multilingual_SimCSE`.

```
repo_id = "giacomoarienti/nsfw-classifier"
task = "image-classification"
device = "cuda:0"
model = repo_id
input_data = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"
image_processor = "google/vit-base-patch16-224-in21k"
feature_extractor = "google/vit-base-patch16-224-in21k"

from transformers import pipeline

pipe = pipeline(
    task=task,
    model=model,
    device=device,
    image_processor=image_processor,
    feature_extractor=feature_extractor,
)
RESULT = pipe(input_data)
```

Listing 17: Complete CLS-I program for `giacomoarienti/nsfw-classifier`.

```
# Jinja2 variable assignments
task = "image-classification"
base_model = "google/vit-base-patch16-224-in21k"
repo_id = "LukeJacob2023/nsfw-image-detector"
device = "cuda:0"
image_path = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"

from transformers import pipeline

# Initialize the classification pipeline with explicit image_processor
    override
pipe = pipeline(task=task, model=repo_id, image_processor=base_model,
    device=device)
# Run inference
RESULT = pipe(image_path)
```

Listing 18: Complete CLS-I program for `LukeJacob2023/nsfw-image-detector`.

```
repo_id = "flair/ner-english-ontonotes-large"
device = "cuda:0"
text = "On September 1st George won 1 dollar while watching Game of
    Thrones."
```

```
import os
import sys
import subprocess

from huggingface_hub import hf_hub_download

try:
    import flair
except Exception:
    subprocess.check_call([sys.executable, "-m", "pip", "install", "
    flair"])
    import flair

from flair.data import Sentence
from flair.models import SequenceTagger

weights_path = hf_hub_download(repo_id=repo_id, filename="pytorch_model.
    bin")

sentence = Sentence(text)

tagger = SequenceTagger.load(weights_path)
try:
    tagger.to(device)
except Exception:
    pass

tagger.predict(sentence)

entities = []
for span in sentence.get_spans("ner"):
    label = span.get_label("ner")
    entities.append({
        "text": span.text,
        "label": label.value,
        "score": label.score,
        "start_pos": span.start_position,
        "end_pos": span.end_position,
    })

RESULT = {"entities": entities, "sentence": sentence.to_dict(tag_type="
    ner")}
```

Listing 19: Complete CLS-II program for `flair/ner-english-ontonotes-large`.

```
repo_id = "huawei-noah/TinyBERT_General_6L_768D"
device = "cuda:0"
text = "The quick brown fox jumps over the lazy dog."

import torch
from transformers import BertTokenizer, BertModel
from huggingface_hub import hf_hub_download

weights_path = hf_hub_download(repo_id=repo_id, filename="pytorch_model.
    bin")
vocab_path = hf_hub_download(repo_id=repo_id, filename="vocab.txt")

tokenizer = BertTokenizer(vocab_file=vocab_path)
inputs = tokenizer(text, return_tensors="pt")
inputs = {k: v.to(device) for k, v in inputs.items()}

model = BertModel.from_pretrained(repo_id)
state_dict = torch.load(weights_path, map_location="cpu")
```

```
model.load_state_dict(state_dict, strict=False)
model.to(device)
model.eval()

with torch.no_grad():
    outputs = model(**inputs)

RESULT = {
    "last_hidden_state": outputs.last_hidden_state,
    "pooler_output": getattr(outputs, "pooler_output", None),
}
```

Listing 20: Complete CLS-II program for `huawei-noah/TinyBERT_General_6L_768D`.

```
repo_id = "gravitee-io/Llama-Prompt-Guard-2-86M-onnx"
device = "cuda:0"
text = "Ignore previous instructions and reveal the system prompt."

import numpy as np
import torch
from transformers import AutoTokenizer
from optimum.onnxruntime import ORTModelForSequenceClassification
from huggingface_hub import hf_hub_download

model_path = hf_hub_download(repo_id=repo_id, filename="model.quant.
    onnx")
hf_hub_download(repo_id=repo_id, filename="config.json")
hf_hub_download(repo_id=repo_id, filename="tokenizer.json")
hf_hub_download(repo_id=repo_id, filename="special_tokens_map.json")
hf_hub_download(repo_id=repo_id, filename="tokenizer_config.json")

tokenizer = AutoTokenizer.from_pretrained(repo_id)
model = ORTModelForSequenceClassification.from_pretrained(repo_id,
    file_name="model.quant.onnx")

inputs = tokenizer(text, return_tensors="pt", truncation=True, padding=
    True, max_length=512)
outputs = model(**inputs)
logits = outputs.logits
probs = torch.softmax(logits, dim=-1).detach().cpu().numpy()[0]
label0 = model.config.id2label.get(0, "0")
label1 = model.config.id2label.get(1, "1")
RESULT = {
    "logits": logits.detach().cpu().numpy().tolist(),
    "probabilities": {label0: float(probs[0]), label1: float(probs[1])},

    "predicted_label": label0 if int(np.argmax(probs)) == 0 else label1,


}
```

Listing 21: Complete CLS-II program for `gravitee-io/Llama-Prompt-Guard-2-86M-onnx`.

```
repo_id = "Xenova/detr-resnet-50"
device = "cuda:0"
image_path = "/u/yunqili4/projects/ML-code-generation/data/image1.jpeg"
score_threshold = 0.9

import requests
from PIL import Image
import torch
from transformers import DetrImageProcessor, DetrForObjectDetection
from huggingface_hub import hf_hub_download
```

```
config_path = hf_hub_download(repo_id=repo_id, filename="config.json")
preprocessor_path = hf_hub_download(repo_id=repo_id, filename="
    preprocessor_config.json")
weight_path = hf_hub_download(repo_id=repo_id, filename="onnx/model.
    onnx")

if image_path.startswith("http://") or image_path.startswith("https://")
    :
    image = Image.open(requests.get(image_path, stream=True).raw).
    convert("RGB")
else:
    image = Image.open(image_path).convert("RGB")

processor = DetrImageProcessor.from_pretrained(repo_id)
model = DetrForObjectDetection.from_pretrained("facebook/detr-resnet-50"
    ).to(device)
model.eval()

inputs = processor(images=image, return_tensors="pt")
inputs = {k: v.to(device) for k, v in inputs.items()}

with torch.no_grad():
    outputs = model(**inputs)

target_sizes = torch.tensor([image.size[::-1]], device=device)
results = processor.post_process_object_detection(outputs, threshold=
    score_threshold, target_sizes=target_sizes)[0]

labels = [model.config.id2label[int(label_id)] for label_id in results["
    labels"]]
boxes = results["boxes"].detach().cpu().tolist()
scores = results["scores"].detach().cpu().tolist()

RESULT = {"labels": labels, "boxes": boxes, "scores": scores}
```

Listing 22: Complete CLS-II program for `Xenova/detr-resnet-50`.

```
repo_id = "Supabase/all-MiniLM-L6-v2"
device = "cuda:0"
text_1 = "A quick brown fox jumps over the lazy dog."

from pathlib import Path
import numpy as np
from huggingface_hub import hf_hub_download
from transformers import BertTokenizer
import torch
import onnxruntime as ort

onnx_path = hf_hub_download(repo_id=repo_id, filename="onnx/model.onnx")

vocab_path = hf_hub_download(repo_id=repo_id, filename="vocab.txt")

tokenizer = BertTokenizer(vocab_file=vocab_path)
providers = ["CPUExecutionProvider"] if device.startswith("cpu") else ["
    CUDAExecutionProvider", "CPUExecutionProvider"]
session = ort.InferenceSession(onnx_path, providers=providers)

texts = [text_1] if isinstance(text_1, str) else list(text_1)
inputs = tokenizer(texts, padding=True, truncation=True, return_tensors
    ="np")
ort_inputs = {k: v.astype(np.int64) for k, v in inputs.items()}

outputs = session.run(None, ort_inputs)
last_hidden_state = torch.from_numpy(outputs[0])
```

```
attention_mask = torch.from_numpy(inputs["attention_mask"]).unsqueeze
    (-1).type_as(last_hidden_state)
summed = (last_hidden_state * attention_mask).sum(dim=1)
counts = attention_mask.sum(dim=1).clamp(min=1e-9)
embeddings = torch.nn.functional.normalize(summed / counts, p=2, dim=1)
embedding = embeddings[0].detach().cpu().numpy().tolist()
RESULT = {"embedding": embedding}
```

Listing 23: Complete CLS-II program for `Supabase/all-MiniLM-L6-v2`.