Run-Time Prevention of Software Integration Failures of Machine Learning APIs

CHENGCHENG WAN, East China Normal University, China
YUHAN LIU, University of Chicago, USA
KUNTAI DU, University of Chicago, USA
HENRY HOFFMANN, University of Chicago, USA
JUNCHEN JIANG, University of Chicago, USA
MICHAEL MAIRE, University of Chicago, USA
SHAN LU, Microsoft / University of Chicago, USA

Due to the under-specified interfaces, developers face challenges in correctly integrating machine learning (ML) APIs in software. Even when the ML API and the software are well designed on their own, the resulting application misbehaves when the API output is incompatible with the software. It is desirable to have an adapter that converts ML API output at runtime to better fit the software need and prevent integration failures.

In this paper, we conduct an empirical study to understand ML API integration problems in real-world applications. Guided by this study, we present SmartGear, a tool that automatically detects and converts mismatching or incorrect ML API output at run time, serving as a middle layer between ML API and software. Our evaluation on a variety of open-source applications shows that SmartGear detects 70% incompatible API outputs and prevents 67% potential integration failures, outperforming alternative solutions.

CCS Concepts: • **Software and its engineering** → Software defect analysis; • **Computing methodologies** → Machine learning; • **Information systems** → RESTful web services.

Additional Key Words and Phrases: software integration failure, machine learning API, run-time patching

ACM Reference Format:
Chengcheng Wan, Yuhan Liu, Kuntai Du, Henry Hoffmann, Junchen Jiang, Michael Maire, and Shan Lu. 2023. Run-Time Prevention of Software Integration Failures of Machine Learning APIs. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 231 (October 2023), 28 pages. https://doi.org/10.1145/3622806

1 INTRODUCTION

1.1 Motivation

Machine learning cloud APIs [Amazon 2022a; Google 2022a; IBM 2022; Microsoft 2022a], referred to as ML APIs in this paper, offer effective solutions for a spectrum of cognitive tasks, relieving programmers from the onerous task of designing, training, and hosting their own machine learning models. Consequently, these APIs are widely used in software applications to solve a variety of

Authors' addresses: Chengcheng Wan, ccwan@sei.ecnu.edu.cn, National Trusted Embedded Software Engineering Technology Research Center, East China Normal University, China; Yuhan Liu, yuhanl@uchicago.edu, Department of Computer Science, University of Chicago, USA; Kuntai Du, kuntai@uchicago.edu, Department of Computer Science, University of Chicago, USA; Henry Hoffmann, hankhoffmann@cs.uchicago.edu, Department of Computer Science, University of Chicago, USA; Junchen Jiang, junchenj@uchicago.edu, Department of Computer Science, University of Chicago, USA, junchenj@uchicago.edu; Michael Maire, mmaire@uchicago.edu, Department of Computer Science, University of Chicago, USA; Shan Lu, shanlu@uchicago.edu, Department of Computer Science, Microsoft / University of Chicago, USA.

2475-1421/2023/10-ART231
https://doi.org/10.1145/3622806

```
1 TARGETS = ['Car','Van','Truck','Boat','Toy vehicle']
2 image = types.Image(content=street_view_image)
3 response = client.object_localization(image=image)
4 for obj in response.localized_object_annotations:
5   if obj.name in TARGETS:
6     predict_flood_depth(obj)
```

Fig. 1. Flood-Depths, a flood detection application [Flood-Depths 2021] using Google Cloud API.

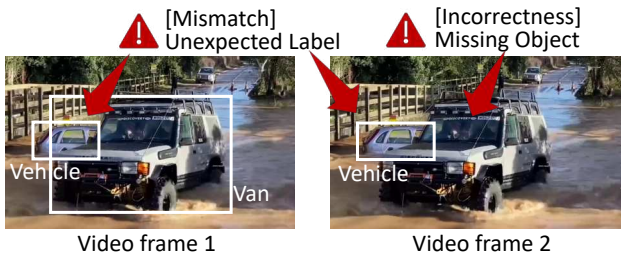

Fig. 2. Two video frames that cause the flood detection application to misbehave.

real-world problems [Das and Behera 2017; Wan et al. 2021]. Unfortunately, these applications often experience *integration failures* in their use of ML APIs, a type of failures that occur when the output of an API invocation is incompatible with the software component that uses this output.

Integration failures are widespread in applications that use ML APIs due to under-specified interfaces of the latter. This phenomenon is particularly visible in two ways. First, every cognitive task performed by an ML API typically has more than one correct answer; unfortunately, state-of-the-art ML APIs do not specify which answer(s) out of the multiple correct ones they would produce. For example, there are many different and correct ways to describe an object, in terms of textile, color, usage, and others. Human beings naturally know what (type of) description matches their conversation and activity context. However, ML APIs cannot guarantee to generate output that fits the expectation of the software , and the software also cannot accommodate the mismatched API output, which leads to integration failures at run time.

Second, the model used by an ML API is inherently probabilistic, whose outputs are only *statistically* reliable over many input samples; unfortunately, state-of-the-art ML APIs do not specify whether or how likely their output for a given input might be incorrect[1]. Without that information, software typically treats the output of an ML API as always correct. This leads to integration failures when the output occasionally is incorrect, which is inevitable for ML APIs.

To better understand these integration challenges and their consequences, consider Flood-Depths [Flood-Depths 2021], an open-source application that uses vehicles as a reference to estimate flood depth. As shown in Figure 1, this application first identifies objects in a `street_view_image` using an ML API. It then iterates through all the identified objects (Line 4), and invokes flood-depth prediction upon every object whose label, stored in the `name` field, matches one of the vehicle-related keywords in the `TARGETS` list, defined on Line 1.

This code snippet seems straightforward on its own and the Google Vision API `object_localization` has a high accuracy. However, they do not work well when integrated together.

[1]Some APIs produce a confidence score to estimate how likely the output is correct. Previous work showed this estimation to be inaccurate, with similar scores produced for correct and incorrect outputs [Bai et al. 2021; Wang et al. 2021a].

Figure 2 shows the API outputs on two adjacent frames in a video. As we can see, the left car is successfully identified by the API in both frames, and is labeled as "vehicle". However, without a clear API specification, software developers do not anticipate the "vehicle" output and do not include it in the `TARGETS` list. As a result of this correct and yet unexpected output ("vehicle"), this car will be mistakenly skipped in the flood-depth prediction. Furthermore, the van in the middle is correctly identified in frame 1, but gets missed in the almost identical frame 2. Again, lacking API specification, the software do not anticipate such incorrect output, and would omit this van in the flood-depth prediction of frame 2. With these vehicles mistakenly skipped, software underestimates or even completely fails to predict flood depth.

As we will see in Section 3, similar problems occur widely in software applications that use ML APIs. The straightforward way of integrating ML APIs, like that in Figure 1, cannot tolerate API results that are out of context or occasionally incorrect, which leads to software misbehavior.

Previous works have not tackled ML API integration failures. They address either problems inside ML APIs [Chen et al. 2022c, 2020b; Xie et al. 2022] or problems inside the software that uses ML APIs [Wan et al. 2021, 2022], but not the integration problems in between. Specifically, several recent work improves the quality of ML software by selecting the most accurate ML API [Chen et al. 2020b] or API ensemble [Chen et al. 2022c; Xie et al. 2022] from several cloud service providers. These techniques only tackle the issues inside the ML component, but do not fundamentally resolve the under-specified ML-API interfaces and hence integration failures. Another line of recent work identifies some ML API misuse patterns [Wan et al. 2022] and offers ways to test ML software [Wan et al. 2021]. The misuse patterns identified so far (e.g., picking the wrong ML API; using higher than necessary input resolution; interpreting floating-point output incorrectly) do not cover integration failures; the testing support offered so far may occasionally expose integration failures under specially designed test inputs, but cannot help detect or reduce integration failures at run time.

1.2 Challenges

It is desirable to have an *adapter* that converts ML API output at run time to better fit the software component that uses the output, compensating for the under-specified interfaces of ML APIs and reducing integration failures. To design such an adapter, there are several challenges.

1) How to judge whether an ML API output matches the expectation of software? Lacking ML API specification, software developers do not know what output they may get from an API invocation and do not have a standard way to specify what output they expect. For example, invoking an object detection API, developers may expect the API output to describe a specific aspect of an object or to fall into a specific set of labels. How to automatically extract that expectation and hence judge whether an API output matches the expectation of software is an open question.

2) How to judge whether an ML API output is correct? Lacking ML API specification, it is difficult to know which specific output of an ML API invocation would be wrong. In fact, judging the correctness of ML APIs that emulate human cognition tasks typically requires human effort, which is difficult to carry out during run-time detection.

3) How to conduct compatibility checking and conversion with low overhead? High overhead is not acceptable for software users. Consequently, we cannot use expensive techniques, such as re-training and updating neural network models at run time, which take hours.

1.3 Contributions

In this paper, we first conduct an empirical study to understand the challenges of integrating machine learning cloud APIs into software and the consequences of integration problems. Specifically, we examine a set of 55 open-source applications that use ML API outputs to make control-flow

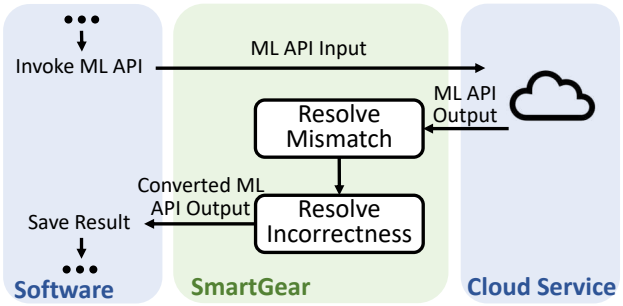

Fig. 3. SmartGear run-time framework.

decisions[2]. We run each application with 100 test inputs and manually check their execution results. We find that on average 15.9% of the test inputs lead to failures. Among these failures, 99.9% of them are non-fail-stop failures related to ML APIs. Since these failures do not throw any exception, they are difficult to detect. *All* of these failures are caused by integration problems: about two-thirds of the failures are caused by correct and yet mismatched ML API outputs, and the other one-third of the failures are caused by incorrect ML API outputs. We further categorize these two types of problems into several sub-types, with details presented in Section 3.

Guided by our study, we propose SmartGear. SmartGear serves as a middle layer between ML APIs and software that uses ML APIs, as illustrated in Figure 3. It reduces the number of ML API integration failures by detecting and converting incompatible ML API output at run time.

To tackle mismatched ML API outputs, SmartGear leverages the insight that ML API outputs which lead the execution towards non-fall-through branch edges are definitely the ones that software expects and knows how to handle, like any `obj.name` leading the program to execute `predict_flood_depth` in Figure 1. We refer to these API-output values as *focal values*. On the other hand, if an API output leads the execution to a fall-through edge, it either is truly not interesting to the software, like a label "Tree" in the flood-detection application, or presents a mismatch with the software, like a label "Vehicle" in the flood-detection application.

To carry out this insight, SmartGear first uses symbolic execution to automatically figure out the sets of *focal values*. Then, at run time, for any API output o that is driving the execution towards a fall-through edge, SmartGear uses a knowledge graph [Wikidata 2022] to compare o with those focal values. Based on the knowledge graph, SmartGear knows whether o is a synonym with some of the focal values, whether o is looking at a different perspective from all the focal values, etc. SmartGear then reports mismatch and conducts output conversion accordingly.

To tackle incorrect ML API outputs, SmartGear leverages an insight about output consistency: the outputs of one ML API upon several similar inputs should be consistent with each other (e.g., object detection results for the two images in Figure 2 should be consistent); the outputs of several related ML APIs upon one input should also be consistent (e.g., if a facial expression API reports a happy face; the object detection API should be able to identify a human face from the same image).

To carry out this insight, SmartGear efficiently compares the outputs of related API invocations at run time and detects inconsistent API outputs. SmartGear resolves such inconsistency and hence reduces potential integration failures by applying ensemble techniques [Dietterich 2000] on these invocations. Furthermore, for ML APIs that produce free-text outputs, SmartGear also checks the consistency between the output o and the focal values. When o only has a minor difference, like

[2]Some applications directly display the output of an ML API. This trivial use of ML APIs is not the target of SmartGear.

one character difference, from a focal value, SmartGear regards *o* as incorrect and converts it to the corresponding focal value.

We evaluate SmartGear using the latest version of 65 open-source Python applications that cover different problem domains and ML APIs. In our evaluation, SmartGear successfully detects 70% of the integration failures and correctly converts 67% of incompatible ML API outputs, improving the correct execution rate of these applications from 84% to 95%.

Our goal in designing SmartGear is not completeness, as catching all failures related to ML techniques is inherently hard. Instead, we offer a first systematic and best-effort attempt to reduce failures caused by ML API integration problems, aligning the software expectation and ML API behavior.

2 BACKGROUND

2.1 ML Cloud Service

ML cloud API. Many cloud service providers [Amazon 2022a; Google 2022a; IBM 2022; Microsoft 2022a] offer ML APIs for vision, natural language, and speech tasks. These tasks cover four categories: (1) classification, which outputs a category that describes an input based on its overall content; (2) recognition, which identifies object/entity/text from an input; (3) synthesis, which generates data from descriptions, *e.g.,* speech synthesis; and (4) translation.

Among these four categories of APIs, classification and recognition APIs are used the most often, contributing to almost 90% of Google and AWS ML API usage in GitHub [Wan et al. 2021]. Consequently, in this paper, SmartGear focuses on all 10 classification and recognition APIs offered by Google, which covers all three domains (vision, language, speech). SmartGear could also be extended to support synthesis and translation tasks, which we leave as future work.

ML API input & output. The input of an ML API typically is an image/video/article/audio together with several configuration parameters. For example, speech APIs require developers to specify the encoding type, sample rate, and language of the input audio. The output of an ML API is usually an array of records, with each record containing several fields that describe an element of the input data and a confidence score. For example, an object detection API outputs a record of object type, bounding box, and a confidence score for each recognized object in the image. To reduce run-time overhead, SmartGear only examines the record fields that are later used by the software.

2.2 Knowledge Graph

A knowledge graph is a database that uses a graph-structured data model to store the description of real-world entities and the relationship between them. Knowledge graphs are widely used for knowledge reasoning in various scenarios, including question-answering systems, search engines, and recommendation systems [Chen et al. 2020a]. Each node in a knowledge graph is called an *entity*, which could be an object, a person, an organization, a location, etc. Typically, each node contains an identifier, a description, a label for the most common name of the item and several aliases. For example, “car” is alias to “motor car”, in Wikidata knowledge base [Wikidata 2022]. Each directed edge between two entities is a *statement* that describes the relationship between two entities or the property of the starting node. For example, an “subclass-of” edge connecting node “van” to node “vehicle” indicates that van is one of the sub-categories of vehicle.

Table 1. Number of applications using different ML Cloud APIs in our benchmark suite

Task		ML Cloud API	# Apps
Vision	Classification	label_detection	14
		web_detection	2
		landmark_detection	1
	Recognition	object_localization	10
		face_detection	3
		text_detection	8
Language	Classification	classify_text	10
	Recognition	analyze_entities	6
		analyze_entity_sentiment	1
Speech	Recognition	recognize[3]	9

3 UNDERSTANDING INTEGRATION FAILURES

3.1 Methodology

3.1.1 Applications. We study 55 Python applications where ML API outputs are used to affect control flow. We focus on this type of applications, as they reflect a non-trivial way of using ML APIs, instead of directly outputting API results. Among these 55 applications, 45 are all the applications in a recently published, real-world ML software benchmark suite [Wan et al. 2022] that use classification and recognition APIs to make control flow decisions. As language classification and speech recognition APIs are the least represented in these 45 applications, we additionally identify 10 more applications on GitHub that use either of these two types of APIs to achieve a more balanced suite of applications.

As shown in Table 1, these applications cover a wide range of machine learning tasks. Their median size is 20,000 lines of code, and 23 of them have received star/fork/watch from GitHub.

For each application, we conduct unit testing upon the entry function that contains the branch whose outcome is directly affected by ML API. For the 6 out of 55 applications that invoke multiple ML API calls, we use these API calls' closest common caller function in the call graph for testing.

3.1.2 Test Data. For each target function to test, we design 100 test inputs to offer roughly even coverage for all paths in the function that invoke an ML API. When there are loops, we consider paths that execute zero or one loop iteration.

To obtain realistic video/image/text inputs for ML APIs, we leverage search engines, Bing search for images/texts and YouTube for videos. For each application, we carefully crafted search keywords that describe the type of inputs related to a path. For example, we feed keywords like "car", "van", and "truck" into Bing image-search to get inputs that execute line 6 in Figure 1. We manually check the top search results and use the ones that we believe are relevant to the software under test. For inputs that potentially contain human identification information, e.g. human faces and vehicle license numbers, we search in public benchmark suites [Kuehne et al. 2011; Panetta et al. 2021] to ensure ethical standards. For audio-processing APIs, since there lacks an audio search engine for real-word audio clips, we designed 45 transcripts based on application scenarios and branch conditions, each containing 1 or 2 English sentences. Seven male and three female participants from authors' institution volunteered to record their speech audios, in their regular way interacting with voice-controlled applications. Participants are between 18 and 30 years old, including both native and non-native English speakers. All tests are independently verified by the two authors.

[3]It includes the synchronous, asynchronous and streaming versions.

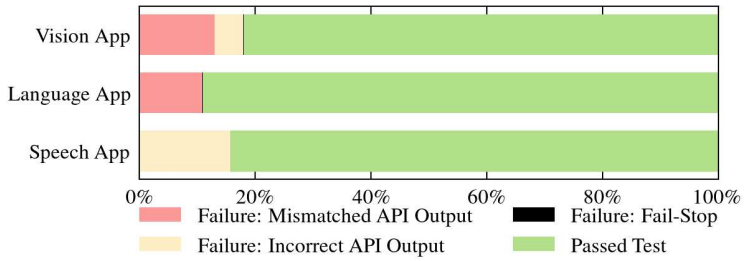

Fig. 4. Testing results for 55 applications.

3.1.3 Failure Identification. We examine each function and decide the correct function behavior for each test input. In two applications, the tested function invokes external library APIs, e.g., flood depth prediction and calorie estimation, which is out of the scope of unit testing. Therefore, we examine whether the parameters of these API calls are correct. Every test case and test oracle is cross-checked by at least two authors.

3.2 Testing Result Summary

Among all the 5500 unit tests (= 100 tests per app ×55 apps), 860 unit tests failed, with an average failure rate of 15.6% and a median failure rate of 8%. The failure rates are similar across applications that use different types of ML APIs (i.e., vision, language, and speech), as shown in Figure 4.

Among these test failures, 99% of them occur in tests that exercise a non-fall-through edge of a branch whose outcome is determined by ML API outputs, like the true edge of the branch on Line 5 of Figure 1. This phenomenon reflects that fall-through edges are much less sensitive to mismatched or incorrect ML API outputs.

At the first glance, one might blame the cloud ML-service provider for high failure rate. However, as shown in Figure 4, for both vision applications and language applications, the majority of testing failures are actually *not* caused by incorrect API outputs. In total, as many as 566 test failures are caused by ML API outputs that are correct but do not match with the software's need.

Among these 860 failures, 4 of them caused the software to crash, and yet the remaining 856 caused silent failures with incorrect software behaviors and yet no exceptions thrown. These silent failures are difficult to automatically detect and are the focus of this paper.

Finally, among all the 55 applications, only 13 applications have not encountered any testing failures, including 5 vision, 6 language, and 2 speech applications. These applications either tolerate mismatched ML API outputs (e.g., some applications created a lookup table for all possible ML API outputs), have a lower accuracy requirement of ML APIs (e.g., some applications simply segment audio inputs by pauses while neglecting the exact transcript), or perform a highly accurate cognition task (e.g. finding nouns in a biography).

3.3 Root Cause 1: Mismatch between ML API and Software

About 65% of failures in our unit testing are caused by correct ML API outputs that are mismatched with the software, just like the mismatched vehicle-label example in Figure 2. As shown in Figure 4, these mismatched ML API outputs are common in both vision applications and language applications, but not in speech applications—speech recognition API could produce incorrect output but not mismatched output, as it outputs audio transcript in free text instead of categorical labels.

Our further study reveals three main types of mismatch.

```
image = types.Image(content=room_photo)
response = client.label_detection(image=image)
predict = response.label_annotations[0].description
category = ""
if predict in ["Light", "Wall plug", "Lighting accessory", "Electrical wiring"]:
  category = "Electrical"
if predict in ["Toilet", "Plumbing", "Sink", "Bathroom", "Washing machine"]:
  category = "Plumbing"
if predict in ["Wall", "Door", "Handle", "Lock"]:
  category = "locksmith"
return category
```

Fig. 5. RoomR, a property management application [RoomR 2020].

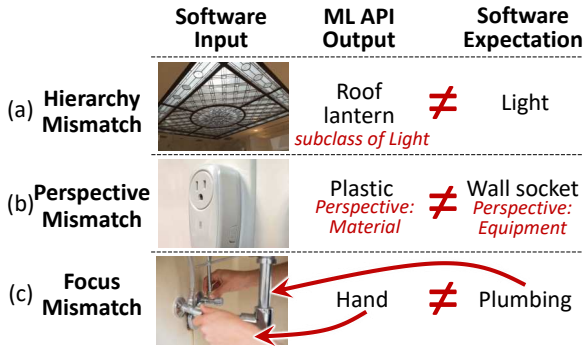

Fig. 6. Three types of mismatch in RoomR (Figure 5).

3.3.1 Hierarchy Mismatch. This is the most common type of mismatch, contributing to about *one-third* of all unit testing failures. Real-world entities have hierarchical relationships, *e.g.,* one is a subclass/component of the other. One object can often be correctly described by multiple labels at different levels of hierarchy. However, a software application may only be coded to recognize one of these labels, causing this hierarchy mismatch. The vehicle problem in Figure 1 is an example.

As another example, the property management application RoomR [RoomR 2020] (Figure 5) uses `label_detection` to classify an indoor photo and extracts the description field of the first API output record. If this field is contained in one of the four pre-defined lists, RoomR maps the photo to the corresponding category. As shown in Figure 6, the ML API outputs "Roof lantern" for a ceiling light photo, which is a correct output for image classification. However, the software misbehaves, failing to put this photo into the "electrical" category. The reason is that the software is coded to recognize "Light" as "electrical", but not its subclass "Roof lantern", as shown in Figure 5.

Besides subclass relationships, hierarchy mismatch also includes the alias relationship. Naturally, people might use different phrases to refer to the same entity. For example, "Light" is under the alias "Light source" and "Light emitter". When the software only examines the alias of an ML API output, it will misbehave even when ML API successfully recognized the corresponding entity.

Hierarchy mismatch is common for two reasons. *First*, ML cloud APIs typically have a large number of categories for classification and recognition tasks, *e.g.,* Google Vision AI has 20,000 category labels with a hierarchical structure. It is hard for developers to exhaustively go through the label set and specify all the related ones. Therefore, ML APIs are very likely to return a mismatched

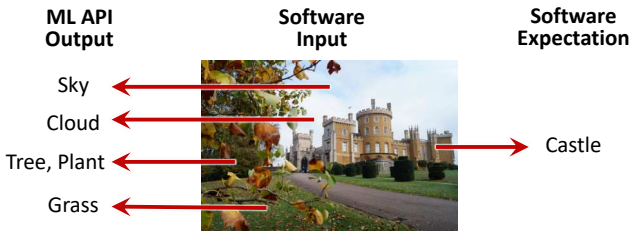

Fig. 8. Focus mismatch in a smart album application [Aander-ETL 2017].

label to the software during run-time. *Second*, the hierarchy of API outputs is greatly affected by its input. In general, the more objects the input includes and the harder the identification is, the more likely the ML API is to output superclass labels to ensure correctness [Wehrmann et al. 2018].

3.3.2 Perspective Mismatch. Given an object, an ML API could output multiple correct labels that describe the object from different perspectives, like shape, texture, color, etc. This typically happens in multi-label classification [Grandini et al. 2020] APIs with a large label set, such as Google's `label_detection` and Amazon's `DetectLabels` APIs. Perspective mismatches contribute to around 3% of all unit testing failures in our study.

Fig. 7. A photo inducing perspective mismatch in a trash classification app [SmartCan 2019].

In the example of Figure 6, the software only examines the equipment type and expects ML API to describe the photo in the same way. However, when given a wall socket photo, the ML API outputs a label describing it by its material ("Plastic"), which is correct but does not match the perspective of the software. As another example, a trash classification application [SmartCan 2019] examines the material of input images. However, when given a photo of a glass ornament (Figure 7), the ML API outputs labels that describe the object in the photo from the perspectives of geometric shape ("Triangle") and color ("Purple"), which are totally different from the perspective of the software.

3.3.3 Focus Mismatch. Sometimes, the software application cares about a particular object in the input image or video, and yet the ML API outputs are about other objects in the input. This contributes to around a quarter of all unit testing failures in our study.

Focus mismatch typically happens to an input that contains many different elements or objects, and the one the software cares about is unfortunately not considered a significant piece of information in the input. The labels of these less dominant elements have a lower rank and thus are ignored by the ML API. In Figure 6, the ML API recognizes hands in the shower faucet photo, as it occupies a large area in the image. Unfortunately, the software expects the API to output "Plumbing" for the shower control valve. As another example, a smart album application [Aander-ETL 2017] wants to find buildings and is given an image of a castle in the countryside, but the castle takes up a relatively small portion of the image (Figure 8). A focus mismatch occurs when the top labels from the ML API are sky, tree, and grass, instead of the castle.

Summary. Mismatched ML API outputs bring new challenges to the reliability of ML applications, as they cannot be eliminated by only inspecting and improving the accuracy of ML APIs.

Fig. 9. Failures in a real-time game tracker for Fortnite Battle Royal [FortniteTracker 2019].

```
audio = RecognitionAudio(content=audio_content)
result = client.recognize(config, audio)[0]
script = result.alternative[0].transcript
if "go to sleep" in script:
  sleep_mode()
elif "play the song" in script:
  play_song()
elif "what is the weather today" in script:
  report_weather()
```

Fig. 10. Lisa-Assistant [Lisa-Assistant 2021], a voice assistant application.

3.4 Root Cause 2: Incorrectness of ML API Output

ML APIs adopt statistical models without strict correctness guarantees. Therefore, from time to time, ML APIs produce incorrect outputs, which lead to about one-third of all the test failures. In the studied applications, incorrect ML API outputs mainly come from two sources.

3.4.1 Neural Network Vulnerability. Due to the probabilistic nature of neural networks, a small perturbation of input might trigger inner flaws of neural network models and leads to an incorrect answer [Szegedy et al. 2014]. This type of incorrectness is likely to be eliminated with a slightly different software input. As shown in Figure 2, while the two video frames have hardly perceptible differences, the `object_detection` API recognizes different numbers of vehicles in them.

As another example, a real-time game tracker [FortniteTracker 2019] uses `text_detection` to extract status logs from screenshots. It then identifies the murderer, means, weapon, and victim from each log line. Figure 9 shows the logging area of three adjacent frames from a game video, which share the same text string at the same position. However, due to minor background changes, the API recognizes different text from them, leading the software to make different decisions.

3.4.2 Low Input Quality. Sometimes, the software input does not contain enough information for ML APIs to perform its cognition task, *e.g.*, a blurry photo or a noisy audio clip. It typically leads to a slightly different output from the ground-truth. Figure 10 illustrates a voice assistant application. It records the user's voice command and uses the `recognize` API to transcribe speech into text strings. It is supposed to play music when the user says "play the song." However, the ML API often wrongly recognizes "play song", missing "the", from audio clips with a high speaking rate. This small mistake causes the software to misbehave, not playing music when it should.

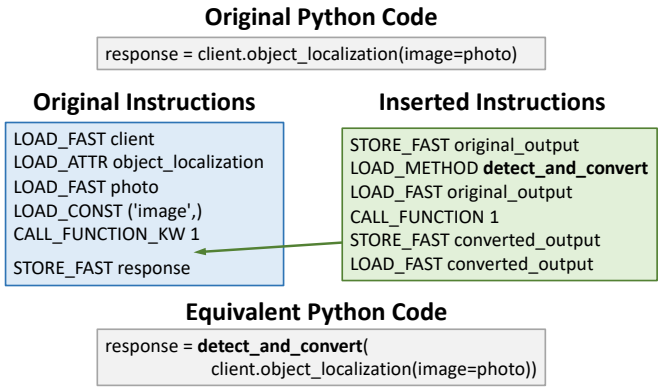

Fig. 11. SmartGear conducts compatibility checking and fixing through byte-code level instrumentation.

Table 2. SmartGear's strategies for detecting and preventing integration failures

Root Cause	Detection	Conversion
Mismatch between ML API and software	Cross checking ML API output and software focal values	ML API output clustering; API input segmentation
Incorrectness of ML API output	Validation across inputs	Ensemble; Video API
	Validation across APIs	Ensemble
	Validation across API and SW	ML API output clustering

Summary. While much prior work focuses on improving neural network accuracy, our study shows that software provides additional information to tackle the incorrectness of ML API. Sometimes, an ML API is invoked on similar inputs, like a series of screenshots, where an incorrect output can be detected by checking output consistency across inputs like the example in Figure 9. Sometimes, the API output is incorrect but very similar to a focal value in the software (e.g., "play song" versus "play the song"), which could have been handled and mitigated. In section 6, we will attempt to tackle incorrect ML API outputs by utilizing such software information.

4 OVERVIEW OF SMARTGEAR

SmartGear is a runtime tool that transparently and automatically checks and converts the output of ML API calls, preventing ML API integration failures.

To use SmartGear, application developers apply a static code instrumentation routine to their applications, which inserts a SmartGear method after every ML API invocation, as shown in Figure 11. At run time, for every ML API output, which is typically an array of values, SmartGear applies a series of incompatibility checking, starting from mismatch checking and followed by incorrectness checking, as listed in Table 2. For every output value o that SmartGear considers to be incompatible, SmartGear generates a warning message and attempts to convert o. Different conversion strategies are designed for different incompatibility situations, as listed in Table 2. When more than one conversion strategy is designed for an incompatibility situation, SmartGear attempts the strategy that incurs the least overhead first. Whenever SmartGear figures out a way to convert an output value o to o', the new value o' is inserted to the original output array, as illustrated in Figure 11.

In the next two sections, we will explain how SmartGear detects and converts mismatched (Section 5) and incorrect ML API output (Section 6).

5 TACKLING MISMATCHED ML API OUTPUTS

At a high level, SmartGear tackles the mismatch between ML API output and software by examining the cognitive relationship between API output and software focal values. It includes several steps.

First, for each invocation I of an ML API, SmartGear conducts static analysis to obtain sets of outputs of this API, with each set driving the execution to cover some non-fall-through branch edge(s) after I. As discussed earlier, these are referred to as focal values and they represent API outputs that the software is interested in and knows how to handle.

Next, at run time, right after the execution of I, SmartGear compares the API output produced there, which is typically an array of values, with focal values associated with I. If at least one of the output values is a focal value, SmartGear assumes that the output is compatible with the software and the execution moves on. Otherwise, SmartGear launches its incompatibility detection.

SmartGear's incompatibility detection checks if there exists hierarchy, perspective, or focus mismatch one by one, guided by the study in Section 3.3. Also guided by the above study, SmartGear's checking is centered on using a knowledge graph to understand cognitive relationships between API output and focal values: Is one the superclass or alias of the other? Is one from a different perspective from the other? Is one related to the other even though there is no superclass relationship?

Once a mismatch is detected, a conversion attempt follows.

In the remainder of this section, we first present how we use static analysis to identify focal values, and then discuss how each type of mismatch is checked in detail.

5.1 Obtaining Focal Values

What are focal values? In many cases, the output of an ML API affects control flow in a straightforward way. For example, in the Flood-Depths code snippet shown in Figure 1, ML API output is used to decide the outcome of the flood-depth prediction branch on Line 5. In this example, output values {`Car`, `Van`, `Truck`, `Boat`, `Toy vehicle`} form a set of focal values, driving the execution towards the non-fall-through edge of the branch.

In some applications, the output of an ML API affects the outcome of multiple branches $\mathbb{B}$, like that in Figure 5. In such a case, more than one set of focal values exists. Every set of focal values drives the execution towards a unique combination of outcomes of $\mathbb{B}$, with at least one branch B, $B \in \mathbb{B}$, taking a non-fall-through edge. We will refer to the unique branch outcome combination associated with each set of focal values as a *focal path*. For example, in the RoomR example, there are three unique sets of focal values, covering three different focal paths.

Finally, very rarely, the software might perform an operation under a branch condition that the ML API output values should not contain or not equal to certain values $\mathbb{V}$. Strictly speaking, this is not a fall-through edge. However, SmartGear still treats it as a fall-through and treats the small set of values the branch condition compares with $\mathbb{V}$ as focal values. The rationale is that the small number of values that software explicitly specifies are the ones that software knows how to handle.

How to identify focal values? To compute a set of focal values for each combination of branch edges affected by ML API, SmartGear adopts a dynamic symbolic execution approach [Irlbeck et al. 2015]. It treats ML API output values as symbolic variables. SmartGear ignores the branches that are unrelated to ML APIs and reconstructs the remaining ones in the format of sub-conditions concatenated with logical OR operators (*e.g.,* `Car or Van or Truck or Boat or Toy vehicle` in Figure 1). SmartGear then generates constraints for all combinations of these sub-conditions of its branch conditions and uses a standard constraint solver to find all the satisfying values of the ML API output, which are the focal values.

5.2 Understanding Cognitive Relationship

We use knowledge graph to understand the cognitive relationship between two cognitive descriptions. Particularly, we want to understand four relationships: subclass/superclass, alias, same/different perspective, and correlation.

For vision tasks, SmartGear leverages a public knowledge graph database Wikidata [Wikidata 2022] to understand the hierarchy, perspective, and other relationships between ML API outputs and software focal values. It is a directed graph, where each node represents a real-world entity, and edges are notated with statements of the relationship between entities. Wikidata supports thousands of statement types and contains over 95 million entities [Haller et al. 2022], covering all the vision categories of popular ML cloud services [Amazon 2022b; Kuznetsova et al. 2020; Microsoft 2022b]. We use this Wikidata knowledge graph in the following ways:

- Sub/super-class: SmartGear traverses the *subclass-of* edges between knowledge graph entities. Two descriptions, like "vehicle" and "car", have a sub/super-class relationship only when one is reachable from the other through subclass-of edges.
- Alias: SmartGear detects alias relationships by querying the knowledge graph to examine whether two descriptions belong to the same entity.
- Same/different perspective: SmartGear treats knowledge graph entities directly connected to the node "entity" or "object" as perspectives, which are typically abstract concepts, e.g. phenomenons, scientific objects, and natural objects. It recursively accesses the superclass of a description, until it reaches these perspectives. If the perspectives of two descriptions have intersection, it regards them as sharing the same perspective.
- Correlation: SmartGear regards two descriptions as having a strong correlation only when their corresponding entities' distance in the knowledge graph is smaller than 2. For example, a "door" is correlated with a "building", as the latter has a "has part(s)" edge directly connecting with the former.

Note that, since the Wikidata knowledge graph contains more than 100 GB of data, the current prototype of SmartGear accesses this online graph database through Wikidata API and caches recently returned API results locally, instead of downloading the whole graph to the local machine.

For language tasks, SmartGear leverages built-in text class hierarchy from cloud service providers, which typically is a directed tree. For example, Google Natural Language AI [Google 2022b] has 620 topic categories in a four-level tree hierarchy, specifying all the superclasses of each category. We uses this text class hierarchy in the following ways:

- Sub/super-class: SmartGear directly uses this class hierarchy graph to determine it.
- Alias: Text topic classification task does not contain aliases.
- Same/different perspective: All the categories have the same perspective of the topic.
- Correlation: SmartGear regards two categories as correlated if they share the same non-root parent node in the class hierarchy, *e.g.*, "investing" and "insurance" are correlated as both of them are the subclasses of "finance".

5.3 Tackling Hierarchy Mismatch

Among the three types of mismatch, a hierarchy-mismatched API output has the closest cognitive relationship to software focal values, as it indicates that the ML API probably detects the object/-concept software is looking for, but outputs a different label in the right class hierarchy. Therefore, SmartGear examines whether there exists a hierarchy mismatch first.

Using the knowledge graph, SmartGear examines if any focal value has a sub/super-class relationship with any value generated by the API: If there is none, SmartGear considers there is no hierarchy mismatch and moves on to detect other potential mismatches.

Fig. 12. Tackling heirachy mismatch with segmentation in RoomR (Figure 6).

If the API output has a sub/super-class relationship with exactly one set of focal values, SmartGear regards that there is a hierarchy mismatch. SmartGear then takes one such focal value and inserts it into the API output to finish the fixing attempt.

Finally, if the API output has sub/super-class relationship with multiple sets of focal values, SmartGear segments the input of the ML API into several pieces, 4 pieces by default, and then applies the ML API again. For example, for an image input, we segment it into 2×2 smaller images; for a text input, we segment it into 4 text pieces with roughly equal length without breaking sentences; and so on. The rationale is that by reducing the number of elements/objects in each input, the ML API is more likely to produce more concrete descriptions, which will help to pin down the exact focal-value set and hence the control flow the software should follow. After obtaining the ML API output for the new set of inputs, If the new output converges to one set of focal values, SmartGear confirms the hierarchy mismatch, adds such a focal value into the original API output, and finishes the incompatibility checking. Otherwise, SmartGear considers there to be no output mismatch and moves on to check the correctness of the API output.

Take RoomR in Figure 5 as an example. Given the ceiling light image in Figure 6a, the `label_detection` API produces a set of labels, including "Roof lantern", "Fixture", "Wood", "Line", and others. SmartGear observes that none of these values are focal values and the execution is heading towards the fall-through edges of all three branches in Figure 5. Therefore, SmartGear starts its hierarchy mismatch checking and finds that among the three sets of focal values, exactly one set ("Light", "Well plug", "Lighting accessory", "Electrical wiring") contains a label "Light" that has a sub-super/class relationship with "Roof lantern". Therefore, SmartGear adds "Light" into the `response` array at Line 2 of Figure 5. An integration failure got avoided.

However, as shown in Figure 12, when the ML API outputs "furniture" initially, SmartGear would not immediately know how to convert this output, as "Furniture" is a super-class of both "Light" and "Sink", which belong to two different focal value sets, electronic set for the former and plumbing set for the latter. In that case, SmartGear would segment the photo into 4 pieces and try again.

Note that, there is a special case of hierarchy mismatch caused by alias. For example, the software might have a focal value "Motor car", but the ML API may output its alias "Car". Once discovering an alias relationship, SmartGear converts the API output to the corresponding focal value.

5.4 Tackling Perspective Mismatch

If no hierarchy mismatch is reported, SmartGear next examines whether the ML API out describes the input in the same perspective as the software. When there is no intersection between the perspectives of *all* software focal values and ML API output values, SmartGear reports a perspective mismatch. Whether two values follow the same perspective is judged using the knowledge graph, as discussed in Section 5.2.

Fig. 13. Tackling focus mismatch with segmentation in RoomR (Figure 6).

Different from hierarchy mismatch, SmartGear only reports a warning instead of attempting to convert the API output. A perspective mismatch indicates that the ML model behind ML API is not trained to recognize the specific perspective, which the software cares about, for this specific input. Any manipulation of the ML API inputs or outputs cannot enlarge the description spectrum to include more description perspectives. For example, the perspectives of all the focal values in RoomR are about equipment. When an ML API output only uses "Plastic" (material perspective) to describe a wall socket image (Figure 6), SmartGear reports a perspective mismatch warning.

5.5 Tackling Focus Mismatch

When ML API outputs do not have hierarchy or perspective mismatch, SmartGear then investigates whether ML API and software might have focused on different items in the same input.

Focus mismatch is very difficult to detect. For example, in RoomR, when ML API outputs "Hand" while the software is looking for "Plumbing", there are three possible reasons: (1) the image contains both a hand and a plumbing equipment, but the ML API wrongly focuses on the hand; (2) the image only contains a hand, and the ML API's output is correct; or (3) the image does not contain a hand, and the ML API makes a completely wrong prediction. While the last case is rare due to the statistically high accuracy of ML APIs, it is hard to distinguish between the first two.

SmartGear examines a focus mismatch in three steps. It *first* confirms there is no hierarchy or perspective mismatch. It *then* examines whether the ML API results and some of the focal values have correlations and hence are likely to appear together in the same scene in real life. If that is the case, SmartGear *finally* applies a segment-and-ensemble strategy to check if the correlated focal value actually exists in the software input.

The way that SmartGear conducts segment-and-ensemble is similar to that in perspective mismatch checking: the input is segmented into 4 pieces with roughly the same size, the ML API is applied to each of the 4 segments, the 4 sets of output is aggregated to form the new output for SmartGear to re-check. If the new output contains a focal value, SmartGear judges the original output to contain a focus mismatch. Otherwise, SmartGear considers the API output to contain no mismatch with the software and moves on to incorrectness checking, described in the next section.

The rationale behind this strategy is that, as discussed in Section 3.3, focus mismatch is caused by the characteristic of software input that the element software cares about is overwhelmed by others. Therefore, after the segmentation, with fewer elements in each API invocation, the ML API may be able to recognize those previously overwhelmed objects in the input. Of course, segment-and-ensemble is relatively expensive, requiring several extra ML API invocations. Therefore, SmartGear carries it out only after several steps of filtering, as discussed above.

Figure 13 shows how SmartGear resolves focus mismatch in RoomR. While the software examines the existence of "Door", the initial output of the ML API only contains "Building" and "Floor" for the door image. SmartGear regards it as a potential focus mismatch, as "Door" and "Building" have a strong correlation, according to the knowledge graph Therefore, SmartGear invokes the API on the four segments of the input image. The new result does contain "Door". Consequently, SmartGear inserts "Door" into the API output, and eliminates the focus mismatch.

5.6 Limitations and Discussions

When SmartGear converts an API output o to a focal value o' that is a subclass of o, this decision could be wrong in theory. For example, if an object-detection API returns "Food" and SmartGear converts "Food" to the focal value "Apple", this decision might be wrong if the input image actually contains an orange instead of an apple. However, in practice, this situation rarely occurs. The reason is that when "Apple" is a focal value, "Orange" is often also a focal value—either "Orange" is in a different focal-value set, where the software is trying to differentiate between different fruits; or "Orange" is in the same focal-value set as "Apple", where the software is trying to differentiate fruit from other types of objects. In the former case, "Fruit" would match with multiple sets of focal values, which would cause SmartGear to apply segment-and-ensemble strategy, instead of randomly converting "Fruit" to one set. In the latter case, converting "Fruit" to either "Apple" or "Orange" makes no difference. We will experimentally evaluate this in Section 8.

SmartGear might make a wrong judgment of perspective mismatch, when the software input is completely unrelated to the software and thus could not be described in the same perspective.

Finally, techniques discussed in this section apply to ML APIs that produce categorical output, which covers all the classification and some of the recognition APIs, like object detection, offered by ML service providers like Google, Amazon, and Microsoft. Techniques in this section do not apply to ML APIs that produce free-text output, while techniques in the next section do apply.

6 TACKLING INCORRECT ML API OUTPUTS

As discussed in Section 1, it is difficult to automatically judge whether the output of an ML API is correct. SmartGear tackles this challenge by leveraging information beyond a single API invocation I: (1) invocations of other APIs on the same input before I in the same run; (2) invocations of the same API on similar inputs before I in the same run; and (3) software focal values. Of course, the first two pieces of information may not exist at every ML API invocation.

6.1 Validation across APIs

Cloud services provide a wide spectrum of ML APIs, and many of them have functional overlaps. Specifically, Table 3 shows the 7 groups of APIs with functional overlaps that we have identified based on the official Google API documentation. SmartGear leverages this functional overlap to detect incorrect API output.

To conduct this detection, SmartGear maintains a global history queue that records the input and output of each ML API invocation. After the invocation I of each ML API A, SmartGear checks the history queue to see if any ML API with functional overlap with A has been invoked in the past upon the same input[4]. When such an invocation I' is identified, SmartGear checks if the output of I is consistent with that of I'. Specifically, using the focal value information, SmartGear checks whether the execution control flow would change if the output of I is replaced with that of I'. When an inconsistency is detected, SmartGear tries to identify which one is incorrect, which we elaborate on in the remaining part of this sub-section. If the output of the latest invocation I is incorrect,

[4]In practice, this occurs quite often, applying to 5 out of 6 applications that use multiple APIs in our benchmark suite.

Table 3. Google ML Cloud APIs with functional overlaps. (Bold ones are more specialized tasks.)

ML Cloud API Groups	Functional Overlaps
label_detection, **object_localization**	Object in image
label_detection, object_detection, **face_detection**	Human in image
label_detection, **landmark_detection**	Landmark in image
web_detection, **label_detection**	Image content
web_detection, **logo_detection**	Logo in image
text_detection, document_text_detection	Text in image
analyze_entities, analyze_entity_sentiment	Proper nouns in article

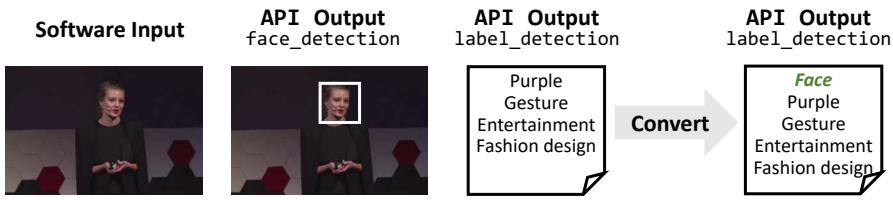

Fig. 14. Resolving inconsistency between APIs in a smart album application [FESMKMITL 2021].

SmartGear conducts output conversion. If the output of the previous invocation I' is incorrect, unable to rollback the history, SmartGear only generates a warning.

SmartGear decides which API's output is correct in two ways depending on the relationship between these two APIs. Sometimes, one API conducts a more specialized task than the other API. Consequently, SmartGear would trust the more specialized API. The reason is that the specialized API typically has higher accuracy on the overlapped task [Yanai and Kawano 2015]. In Table 3, the bold-font APIs are the more specialized ones in their corresponding groups. In other cases, if the two APIs both conduct specialized tasks, SmartGear trusts the one that contains more focal values.

For example, a smart album application [SeeFarBeyond 2022] uses Google's `face_detection` API to find human faces in the image. It then invokes Google's `label_detection` API to categorize the same image, with the focal value "Face". While the former is a more specialized API for finding humans in the image, the latter API is also able to classify human images. Therefore, when `face_detection` API recognizes a face but `label_detection` does not contain this focal value, SmartGear regards the latter as incorrect and adds "Faces" to its output array (Figure 14).

6.2 Validation across Inputs

As discussed in Section 3.4.1, a small change of the input sometimes triggers inner flaws of neural network models and leads to an incorrect result, as shown in Figure 2 (frame 2) and Figure 9. SmartGear detects an incorrect API output o if o is inconsistent with the outputs recently produced by the same API upon similar inputs.

Specifically, for each ML API invocation site, SmartGear maintains a history queue that records the last K inputs and outputs processed and produced at this site (K is configurable and set to 5 by default). Upon every ML API invocation that produced an output o based on an input i, SmartGear searches the corresponding history queue for any past input i' that is similar with i, and checks whether the output of i', denoted as o', is consistent with o. We consider o and o' to be consistent, if they are expected to influence software control flow in the same way, which SmartGear judges based on the focal value information. To minimize performance overhead, SmartGear uses light-weighted algorithms to measure input similarity: perceptual hash values are used for image inputs

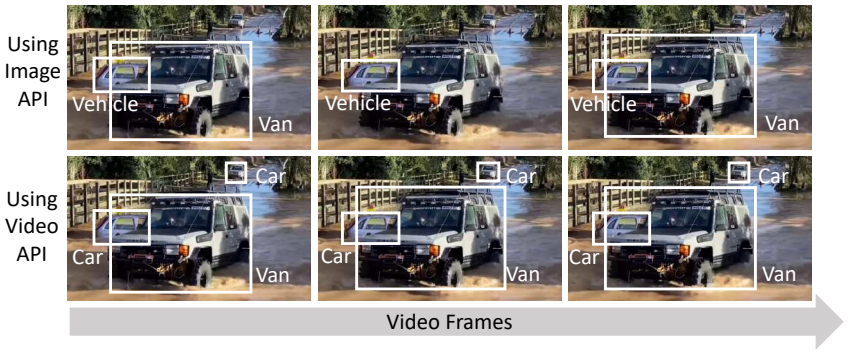

Fig. 15. Applying image and video APIs to the input of Flood-Depth (Figure 1).

and Levenshtein distance is used for text inputs. SmartGear does not measure audio similarity, as it is very rare for an application to process two input audio clips that are similar to each other.

After detecting inconsistencies, SmartGear tries to convert the output of the latest invocation based on the history. If the software takes images as input, SmartGear utilizes temporal information by concatenating several recent images, including the latest one, into a video and using a video API to get a more accurate result, which is then used to convert the original image API output. For non-vision APIs, SmartGear uses the majority voting rule to figure out a value to convert the latest API output. As shown in Figure 15, SmartGear detects that the `object_localization` API provides different vehicle detection results across adjacent video frames. SmartGear then invokes the corresponding video API, which finds all the vehicles in the video, and forwards the last frame result to the software as a correction of the most recent invocation of the original vision API.

6.3 Validating across Software and ML API

There is an additional chance to detect incorrect output for recognition tasks whose outputs are free text, instead of pre-defined labels. Examples of these tasks include image-to-text, speech-to-text, and others. These tasks typically adopt RNN or LSTM and perform a scanning strategy to recognize text from images and audio [Messina and Louradour 2015; Shewalkar 2019]. Therefore, when they make mistakes, these mistakes tend to be small and local, e.g. missing a character.

SmartGear checks whether *any* ML API result could become a software focal value with edits (Levenshtein distance) less than a threshold portion (by default 30%) of the average length of the shortest focal string of each execution path. If there is only one related path, SmartGear clusters ML API results to it and converts its value to the corresponding focal string. In the Lisa-Assistant example (Figure 10), SmartGear finds that API output "play song" could become "play the song" by editing one word, while having a much larger distance to other branches' focal values (*e.g.*, "what is the weather today"). Therefore, it clusters API output to the song branch.

If there are multiple related paths, SmartGear reports a warning that the software is not able to distinguish them. The voice-activated light application (Figure 16) is an example. It records the user's voice command and uses `recognize` API to transcribe speech into text strings. It will turn on/off the light with the command "light on"/"light off". Tested with an inarticulate audio, the ML API wrongly recognizes "light on" as "nike on". While SmartGear finds that "right on" has the smallest edit distance to the focal values of the turn-on branch, it also has a similar edit distance to the turn-off branch. In this case, SmartGear reports a warning without converting API outputs.

```
audio = RecognitionAudio(content=audio_content)
response = client.recognize(config, audio)
for result in response.results:
  script = result.alternative[0].transcript
  if script=="light on" or script=="lights on":
    turn_on_light()
  if script=="light off" or script=="lights off":
    turn_off_light()
```

Fig. 16. A voice-activated light application [ProjectSyn 2020].

6.4 Limitations and Discussions

SmartGear might raise false alarms when the software is sensitive to minor differences of ML API outputs, *e.g.,* the audio transcript must be in the singular form of a certain word. SmartGear also assumes that ML API is reliable for most of the time. It cannot prevent failures if the machine learning model has a low overall accuracy on software inputs. SmartGear is only able to convert the output of the most recent ML API invocation. cannot rollback to convert it. Finally, when SmartGear uses focal values to validate API output, it cannot handle output that already incorrectly contains focal values.

7 IMPLEMENTATION

The SmartGear approach is general and to various ML Cloud services and programming languages. In this paper, SmartGear is implemented for Python applications and Google Cloud AI [Google 2022a], which is the most popular language and ML Cloud services on GitHub [Wan et al. 2021].

SmartGear is implemented as a run-time library with Python function decorator interface [Smith et al. 2022]. To use SmartGear, developers only need to specify the target function, without changing the software implementation. SmartGear then uses Bytecode module [Stinner 2021] and compiler infrastructure to insert detection and conversion instructions between ML API invocation and output storage. SmartGear uses Pyan [Marby and Yonskai 2021] and Jedi [Halter 2022] library, symbolic execution framework PyExZ3 [Irlbeck et al. 2015], and CVC constraint solver [Barrett et al. 2011] for static analysis. SmartGear uses Wikidata python interface [Minhee 2021] for querying knowledge graphs and ImageHash library [Buchner 2022] for image perceptual hash.

8 EVALUATION

8.1 Methodology

8.1.1 Applications and Test Data. We evaluate SmartGear using 65 applications. These include two sets: 1) all the 55 applications in our empirical study (Section 3.1.1) and 2) 10 additional applications sampled *after* the empirical study and the design of SmartGear. To identify these 10 applications, we used a GitHub API to crawl and obtain a list of 300 top-ranked GitHub applications that invoke ML APIs. We then manually examined these 300 applications and filtered out those that do not use ML API output for control flow decisions and those already in our benchmark suite. Finally, we took the top 2 remaining applications for each of the five task types listed in Table 1. The two sets of applications turn out to have the same average ages (i.e., 35 months old as of April 1st, 2023) and similar sizes (8185 LoC for the 55 applications, and 9924 LoC for the additional 10 applications). Section 8.2.4 compares the evaluation results of these two sets of applications.

We design a *new* set of 100 test cases for each application, following the same methodology used in our empirical study (i.e., the test inputs used in our evaluation do **not** overlap with those used

in our empirical study in Section 3). This set of inputs led to 1026 failed tests encountered by 52 applications, including 4 fail-stop failures. Among the 1022 silent failures without exceptions thrown, 696 are caused by mismatched API outputs and 326 are caused by incorrect API outputs—roughly the same breakdown as the previous set in Section 3.

8.1.2 Metrics. SmartGear has two usage scenarios. *First*, its capability of detecting incompatible ML API outputs could be used to caution app-users and to guide application developers to implement application-specific fixing solutions. *Second*, its capability of converting incompatible ML API outputs can prevent integration failures at the run time.

Therefore, we evaluate both the detection and the prevention capability of SmartGear and baselines. If a technique detects a fail-stop symptom or an incompatible ML API output that leads to software misbehavior in a test, we refer to it as successfully *detecting an error*. If it eliminates software misbehavior by converting API output, we refer to it as successfully *preventing a failure*.

8.1.3 Baselines. Since there is no prior work tackling ML API integration failures at run time, we have designed 4 baseline techniques, as listed below. The first 3 only support error detection, and the last one aims to not only detect but also fix incompatible ML API outputs. As we will see, these baselines all suffer from severe false positive or false negative issues. They are designed only for comparison purposes, instead of realistic deployment.

Crash-Only: It conducts error detection by capturing software crashes, unhandled exceptions, and assertion failures. This baseline does not attempt to convert erroneous API output.

ML-API-Only: In addition to monitoring fail-stop failure symptoms, it also uses the confidence score of ML cloud APIs to detect incorrect ML API outputs. In each execution, it reports an error if the minimum confidence score of an ML API output is lower than a certain threshold. To understand the full potential of this baseline technique, we exhaustively searched the threshold space between 60% to 100%, with an interval of 1%, and identified 72% as the setting that offered the most accurate results for our benchmark. This baseline does not attempt to convert erroneous API output.

Software-Only: In addition to monitoring fail-stop symptoms, it also examines the control flow of software. In each execution, it reports an error whenever the software executes a *non-focal path*. This baseline does not attempt to convert erroneous API output.

Software-Segment: This baseline attempts not only error detection but also failure prevention. For each ML API output o, this technique first applies *Software-Only* to see if o might be erroneous. If so, this technique segments the API's image/text/audio input into four equal-size pieces, applies the ML API to each piece, and then aggregates the output from each piece to form a new output o'. If o' is different from o, an API output error of the original software is reported and o will be converted to o' to prevent potential failures.

8.2 Evaluation Results

For each application in our benchmark suite, we apply SmartGear and four baselines on its 100 test cases. Table 4 shows the overall results.

8.2.1 Incompatible ML API Output Detection. As shown in Table 4, across 6500 testing runs, SmartGear reports 744 errors, with 718 true positives (i.e., the corresponding API output is indeed mismatching or incorrect) and only 26 false positives. The 718 true ML API output errors detected by SmartGear constitute 70% of all the 1026 ML API output errors that occurred during the 6500 testing runs.

For the 308 incompatible ML API output missed by SmartGear, there are mainly two reasons. First, about 60% of false negatives are focus mismatches, which are not only common but also inherently difficult to detect, as discussed in Section 5.5. Second, over a third of these false negatives

Table 4. Result summary across 6500 testing runs, 100 each across 65 apps. (True Positive: a correct detection, where the reported API output is indeed incorrect or mis-matching; False Positive: an incorrect detection, where the reported API output is correct and matching.)

	Detection		Prevention	
	# True Positives	# False Positives	# Execution Failures	Execution Correctness Rate
No Tool	-	-	1026	84.2%
SmartGear	718	26	349	94.6%
Crash-Only	4	0	-	-
ML-API-Only	154	162	-	-
SW-Only	905	1050	-	-
SW-Seg	140	0	890	86.3%

are incorrect API outputs, SmartGear fails to detect them as SmartGear assumes that ML API is reliable for most of the time, a limitation discussed in Section 6.4.

Among the 26 false positives in SmartGear's detection, all but one of them occur during the mismatch detection (Section 5). Particularly, SmartGear incorrectly reported a number of perspective mismatches for inputs that are somewhat irrelevant to the software under test. For example, a plant management application [Plant-Watcher 2018] checks if an image is about "Plant" (natural object) or "Flowerpot" (production good). Given a somewhat irrelevant city sky-view picture, the `label_detection` API correctly outputs "Building" (artificial entity) and "Urban design" (research object), and the application correctly considers the picture as neither plant nor flowerpot. Unfortunately, SmartGear wrongly reports a perspective mismatch, as there is no overlap between the perspectives of ML API outputs and software focal values, which are shown in the parentheses.

In comparison, the other four schemes have much lower detection coverage (i.e., many more false negatives) or much lower detection accuracy (i.e., many more false positives) or both.

In terms of detection coverage, *Crash-Only*, *ML-API-Only*, and *SW-Seg* only detect 4, 154, and 140 API errors respectively, less than a quarter of what SmartGear detects. *Crash-Only* naturally has the least coverage, as it can only detect fail-stop symptoms. *ML-API-only* performs poorly because the confidence score of an ML API cannot be used to tell whether the API output is compatible or not with the software context. In fact, our experiment shows that compatible ML API output and incompatible output have similar confidence score distributions: across all ML API invocations in test cases, the confidence scores of compatible (incompatible) output range from 54%(50%) to 96%(94%), with 80% (79%) being the median. *SW-Seg* also has many false negatives, because its segment-and-aggregate technique is helpful for detecting focus-mismatch output but not other types of incompatible output. Take the code snippet in Figure 5 and the image in Figure 6a as an example. SmartGear successfully detects a hierarchy mismatch between "roof lantern" and "light". This error is missed by *Crash-Only*, as no crash or exception occurred; this is also missed by *ML-API-Only* as this particular API invocation has a confidence score of 75%, higher than the 72% threshold; as the ML API does not output "light" on the segmented images, *SW-Seg* also fails.

In terms of detection accuracy, *ML-API-Only* and *SW-Only* report more than 5× and 40× false positives than SmartGear. The inaccuracy of *ML-API-Only* is because confidence scores are not indicative of the incompatibility between an ML API output and the software context, as discussed above. The inaccuracy of *SW-Only* is caused by its aggressive error-detection strategy: by reporting an error *whenever* the software executes a *non-focal path*, *SW-Only* reports more false positives than true errors (i.e., 1050 vs. 905 as shown in Table 4). Comparing with SmartGear, *SW-Only*

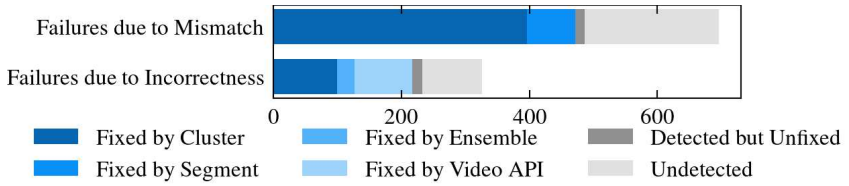

Fig. 17. The number of original testing failures that can be fixed by different SmartGear strategies.

reports 1.26× more true errors, but at the cost of 40× more false positives. Its detection strategy also offers no hint for output conversion or failure prevention, unlike SmartGear.

8.2.2 Incompatible ML API Output Conversion. The ultimate goal of SmartGear is to improve applications' execution correctness rate. As shown in the last column of Table 4, SmartGear achieves this goal by improving the average execution correctness rate across all 65 applications from 84.2% to 94.6%. Specifically, for the 744 API outputs that SmartGear believes to be incompatible, it comes up with fixing strategies for 726 of them. Since SmartGear's detection incurs a small number of false positives, 9 of these fixing attempts turn originally correct API outputs to be incorrect. Fortunately, the remaining 717 fixing attempts target truly incompatible API outputs, and turn originally incompatible API outputs to be correct and compatible in 686 cases. As a net result, SmartGear reduces the number of failed testing runs from 1026 down to 349.

Figure 17 shows the effectiveness of each of SmartGear's fixing strategies. SmartGear resolves 68% mismatch between ML API and software with clustering and segmentation. For incorrect output, SmartGear resolves 67% of them with clustering, ensemble, and video APIs.

In comparison, *Software-Segment*, the other scheme that has failure prevention capability, is only able to eliminate 126 of 1026 failures, improving execution correctness by only 2%.

8.2.3 Ablation Study. To understand the effect of each heuristic in SmartGear, we conduct an ablation study shown in Table 5. In (A)-(F), we each remove one of the heuristics introduced in Section 5&6.

(A) and (C) show that detecting and resolving hierarchy and focus mismatch contribute the most to the result of SmartGear. In other words, the detection coverage and the failure-prevention capability of SmartGear drop the most when we remove these two strategies. This reflects the fact that hierarchy mismatch and focus mismatch are the most common types of incompatible API output, as discussed in Section 3.3. Furthermore, the result of (A) also reflects the fact that SmartGear is particularly effective in detecting hierarchy mismatch: 99% of hierarchy mismatch in testing runs are correctly detected by SmartGear.

The result of (B) indicates that tackling perspective mismatch has the least impact on the overall result of SmartGear. This small impact is due to two reasons. First, perspective mismatch is the rarest type of incompatible API output, as also discussed in Section 3.3. In this experiment, it only contributes to 23 unit test failures (out of the total 1026 unit test failures). Second, to avoid false positives, SmartGear uses a conservative approach to reporting perspective mismatch. As a result, SmartGear incurs no false positives in its detection of perspective mismatch but only manages to detect 4 out of these 23 perspective mismatch errors. How to improve the detection coverage for perspective mismatch without sacrificing the detection accuracy is a research topic for future work.

The strategies behind (D)–(F) are used by SmartGear to detect and fix incorrect ML API output. As displayed in (D)–(F), these three validating strategies all provide non-negligible contributions to SmartGear, each detecting 37 to 100 incorrect API output and preventing 27 to 98 failures.

Table 5. Ablation study of SmartGear over each heuristic (Total # of original test failures is 1026 (100%))

	#(%) Detected Errors	#(%) Prevented Failures
SmartGear	718 (70%)	686 (67%)
(A) Remove resolving hierarchy mismatch (Section 5.3)	389 (38%)	357 (35%)
(B) Remove resolving perspective mismatch (Section 5.4)	714 (70%)	686 (67%)
(C) Remove resolving focus mismatch (Section 5.5)	547 (53%)	544 (53%)
(D) Remove validating across APIs (Section 6.1)	681 (66%)	659 (64%)
(E) Remove validating across inputs (Section 6.2)	624 (61%)	596 (58%)
(F) Remove validating across API and SW (Section 6.3)	618 (60%)	588 (57%)

8.2.4 Sensitivity across Apps. SmartGear shows similar detection and prevention capability over two application sets, the 55 applications used in our empirical study and the 10 additional applications. Specifically, SmartGear detects 642 (70%) API output errors and prevents 623 (68%) out of 920 testing failures in the original 55 applications; it detects 76 (71%) errors and prevents 63 (60%) out of 106 testing failures in the additional 10 applications. SmartGear improves the execution correctness rate from 83% to 94% for those 55 applications; and from 89% to 96% for the additional 10 applications.

8.2.5 Performance Overhead. Due to computation workload and network transmission, ML API invocations typically take half to several seconds [Wan et al. 2021], much longer than other computations in integration failure prevention. Therefore, the run-time overhead is mainly caused by extra ML API invocations, e.g., invocation upon segmented input (Section 5.3 & 5.5) and invoking video API (Section 6.2). Note that, SmartGear uses one batch API, instead of four separate API calls, to process the four segmented inputs to reduce overhead. Overall, across all 5500 test cases, SmartGear introduces 6% performance overhead on average. When there is no extra ML API invocation, the overhead of SmartGear is around 1%. In comparison, the *Software-Segment* baseline invokes many more ML APIs, and incurs 40% overhead on average across all test cases.

8.3 Comparison with Non-Runtime-Prevention Approaches

As discussed in Section 1, some related previous work tries to either correct wrong coding in the software or improve the accuracy of ML APIs. In the following, we use our benchmark suite to quantitatively show that prior work does not address integration failures.

Static code analysis: A previous work [Wan et al. 2021] summarizes 3 types of ML API misuses that reduce software functionality. We examine all the applications in our benchmark suite, and only find the existence of misuse pattern *using the wrong API*: wrongly using `text_detection` API to extract handwritten or long-form text from images. We manually patch software by replacing `text_detection` API with `document_text_detection` API. We run the tests in Section 8.2 again and find that these code patches eliminate 4 failures in 3 applications.

Automated testing: A recent work [Wan et al. 2021] automatically generates test cases and suggests implementation changes for ML software. Instead of preventing failures for each individual input at run time, it aims to reveal software implementation problems that have statically significant impacts on the correctness of its synthesized test cases. When applied on our benchmark suite, this technique suggests code changes in 12 applications. We adopted these changes and executed the tests in Section 8.2. The evaluation results show that these code changes only eliminate 51 failure runs in our benchmark suite, improving the execution correctness rate from 84.2% to 85%.

ML API selection: Several works [Chen et al. 2022c, 2020b; Xie et al. 2022] aim at reducing software failures by dynamically selecting several ML APIs from a set of service providers, with a constraint on service costs. These approaches focus on the ML API itself instead of integration

failures. We do not include them as a baseline, as they use ML APIs from different service providers. In fact, even when an ideal 100% accurate ML API is chosen, more than half of failures remain, as they are caused by the mismatch between ML API and software. On the contrary, SmartGear further improves software accuracy from 89% to 97% after the software adopted an ideal ML API.

9 THREATS TO VALIDITY

Internal threats to validity. SmartGear assumes that the software implementation and code structure represent the distribution of its run-time inputs, which could be incorrect. The software may be deployed in an environment different from its designed scenario, where SmartGear cannot cross-check between ML API and software. SmartGear assumes that ML cloud APIs are reliable at most of the time, which is not guaranteed.

External threats to validity. SmartGear is only tested with Python applications using Google Cloud AI, which may not represent all real-world applications. We design application test data using handcrafted search keywords and limited sources, which may be biased and not cover all possible software run-time scenarios.

10 RELATED WORK

We discussed some closely related work in Section 1 and 8.3. Here we discuss other related work.

Recent work studies the security of cloud ML services, including defending against attacks [Hou et al. 2019] and detecting vulnerability [Hosseini et al. 2017; Pajola and Conti 2021]. Other work evaluates the effectiveness of ML cloud service systems [Yao et al. 2017] and ML API shifts [Chen et al. 2022a]. Another work [Chen et al. 2019] studies cloud service cost. They focus on server-side implementation instead of using ML API in software.

Prior work [Amershi et al. 2019; Hill et al. 2016; Kim et al. 2016, 2017; Nahar et al. 2022; Zhao and Gao 2018] studies the principle and challenges for the development team to implement and maintain software that contains machine learning components. They do not provide solutions for integration failure. Another line of work studies testing [Cheng et al. 2018; Helle et al. 2016; Lindvall et al. 2017; Linz 2020; Zhang et al. 2018] and fixing [Wu et al. 2021] algorithms for software using their own ML solutions. These works focus on optimizing ML models to improve overall software accuracy, instead of tackling integration failures for individual inputs.

Recent research [Hendrycks et al. 2019; Jaiswal et al. 2020] explored self-supervised learning, which updates neural network parameters during inference. Some other work [Chen et al. 2022b; Mummadi et al. 2021; Royer and Lampert 2015; Voigtlaender and Leibe 2017; Wang et al. 2021b; Zintgraf et al. 2019] further adopts run-time domain adaptation to generalize neural networks. These work aims to improve the accuracy of neural networks for run-time data that was not seen during training. They are orthogonal to our approach.

Recent work has looked at how to integrate web APIs into software [Aué et al. 2018; Grent et al. 2021; Wittern et al. 2017]. Unlike ML-API integration problems, generic web API integration problems studied in those work all cause crashes or explicit error code returned by API calls.

The research direction of automatic run-time error patching [Lewis and Whitehead 2010; Long et al. 2014; Perkins et al. 2009] has been pioneered by ClearView [Perkins et al. 2009]. ClearView attempts to fix a range of run-time errors, such as memory errors, by automatically changing variable values or control flows to restore program invariants. SmartGear can be viewed as an automatic run-time error patching tool that is tailored for fixing ML API integration errors by changing ML API output.

11 CONCLUSION

Integration failures are widespread and challenging to tackle in machine learning software. In this paper, we study their root causes and present SmartGear, a run-time tool that prevents some integration failures by detecting and converting incompatible ML API output. SmartGear is only a starting point in tackling the integration challenge in using machine learning techniques in software. We hope SmartGear can inspire future research along this direction.

DATA-AVAILABILITY STATEMENT

The artifact that includes our benchmarks, SmartGear source code, emprical study and experimental results is available on Github [Wan et al. 2023b] and Zenodo [Wan et al. 2023a].

ACKNOWLEDGEMENT

We thank the reviewers for their insightful feedback. The authors' research is supported by NSF (CNS1764039, CNS1956180, CCF2119184, CNS1952050, CCF1823032), ARO (W911NF1920321), and a DOE Early Career Award (grant DESC0014195 0003). Additional support comes from the CERES Center for Unstoppable Computing, UChicago Marian and Stuart Rice Research Award, Microsoft research dissertation grant, and research gifts from Facebook.

REFERENCES

Aander-ETL. 2017. A smart album application. Online document https://github.com/Grusinator/Aander-ETL.

Amazon. 2022a. Amazon artificial intelligence service. Online document https://aws.amazon.com/machine-learning/ai-services.

Amazon. 2022b. Amazon Rekognition Image. Online document https://aws.amazon.com/rekognition/image-features/.

Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *ICSE-SEIP*. IEEE, 291–300. https://doi.org/0.1109/ICSE-SEIP.2019.00042

Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. 2018. An exploratory study on faults in web API integration in a large-scale payment company. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 13–22.

Yu Bai, Song Mei, Huan Wang, and Caiming Xiong. 2021. Don't just blame over-parametrization for over-confidence: Theoretical analysis of calibration in binary classification. In *International Conference on Machine Learning*. PMLR, 566–576. https://doi.org/10.48550/arXiv.2102.07856

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Johannes Buchner. 2022. ImageHash: An image hashing library written in Python. Online document https://pypi.org/project/ImageHash/.

Dian Chen, Dequan Wang, Trevor Darrell, and Sayna Ebrahimi. 2022b. Contrastive Test-time Adaptation. In *CVPR*. 295–305. https://doi.org/10.1109/CVPR52688.2022.00039

Lingjiao Chen, Tracy Cai, Matei Zaharia, and James Zou. 2022a. Did the model change? Efficiently assessing machine learning API shifts. In *ICLR Poster*. https://doi.org/10.48550/arXiv.2107.14203

Lingjiao Chen, Paraschos Koutris, and Arun Kumar. 2019. Towards model-based pricing for machine learning in a data marketplace. In *Proceedings of the 2019 International Conference on Management of Data*. 1535–1552. https://doi.org/10.1145/3299869.3300078

Lingjiao Chen, Matei Zaharia, and James Zou. 2022c. FrugalMCT: Efficient Online ML API Selection for Multi-Label Classification Tasks. In *PMLR*. https://doi.org/10.48550/arXiv.2102.09127

Lingjiao Chen, Matei Zaharia, and James Y Zou. 2020b. Frugalml: How to use ml prediction apis more accurately and cheaply. In *Advances in neural information processing systems*, Vol. 33. 10685–10696. https://doi.org/10.48550/arXiv:2006.07512

Xiaojun Chen, Shengbin Jia, and Yang Xiang. 2020a. A review: Knowledge reasoning over knowledge graph. *Expert Systems with Applications* 141 (2020), 112948.

Chih-Hong Cheng, Chung-Hao Huang, and Hirotoshi Yasuoka. 2018. Quantitative projection coverage for testing ml-enabled autonomous systems. In *International Symposium on Automated Technology for Verification and Analysis*. https:

//doi.org/10.48550/arXiv.1805.04333

Kajaree Das and Rabi Narayan Behera. 2017. A survey on machine learning: concept, algorithms and applications. *International Journal of Innovative Research in Computer and Communication Engineering* 5, 2 (2017), 1301–1309.

Thomas G Dietterich. 2000. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*. Springer, 1–15. https://doi.org/10.1109/IDEA49133.2020.9170675

FESMKMITL. 2021. A smart album application. Online document https://github.com/matthewjmc/FESMKMITL.

Flood-Depths. 2021. A flood detection application. Online document https://github.com/nlonberg/flood-depths.

FortniteTracker. 2019. A real time game tracker application. Online document https://github.com/Godsinred/FortniteKillfeed.

Google. 2022a. Google Cloud AI. Online document https://cloud.google.com/products/ai.

Google. 2022b. Google Cloud Natural Language. Online document https://cloud.google.com/natural-language/docs/categories.

Margherita Grandini, Enrico Bagli, and Giorgio Visani. 2020. Metrics for multi-class classification: an overview. *arXiv preprint arXiv:2008.05756* (2020). https://doi.org/10.48550/arXiv.2008.05756

Henk Grent, Aleksei Akimov, and Maurício Aniche. 2021. Automatically identifying parameter constraints in complex Web APIs: A case study at Adyen. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 71–80. https://doi.org/10.48550/arXiv.2102.00871

Armin Haller, Axel Polleres, Daniil Dobriy, Nicolas Ferranti, and Sergio J Rodríguez Méndez. 2022. An Analysis of Links in Wikidata. In *European Semantic Web Conference*. Springer, 21–38. https://doi.org/10.1007/978-3-031-06981-9_2

Dave Halter. 2022. Jedi: an awesome auto-completion, static analysis and refactoring library for Python. Online document https://jedi.readthedocs.io.

Philipp Helle, Wladimir Schamai, and Carsten Strobel. 2016. Testing of autonomous systems–Challenges and current state-of-the-art. In *INCOSE international symposium*. https://doi.org/10.1002/j.2334-5837.2016.00179.x

Dan Hendrycks, Mantas Mazeika, Saurav Kadavath, and Dawn Song. 2019. Using self-supervised learning can improve model robustness and uncertainty. *Advances in neural information processing systems* 32 (2019). https://doi.org/doi/10.5555/3454287.3455690

Charles Hill, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. 2016. Trials and tribulations of developers of intelligent systems: A field study. In *VL/HCC*. https://doi.org/10.1109/VLHCC.2016.7739680

Hossein Hosseini, Baicen Xiao, and Radha Poovendran. 2017. Google's cloud vision api is not robust to noise. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 101–105. https://doi.org/10.48550/arXiv.1704.05051

Jiahui Hou, Jianwei Qian, Yu Wang, Xiang-Yang Li, Haohua Du, and Linlin Chen. 2019. Ml defense: against prediction API threats in cloud-based machine learning service. In *Proceedings of the International Symposium on Quality of Service*. 1–10. https://doi.org/10.1145/3326285.3329042

IBM. 2022. IBM Watson. Online document https://www.ibm.com/watson.

M Irlbeck et al. 2015. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering* 40 (2015), 26. https://doi.org/10.3233/978-1-61499-495-4-26

Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and Fillia Makedon. 2020. A survey on contrastive self-supervised learning. *Technologies* 9, 1 (2020), 2. https://doi.org/10.1109/ICAI55435.2022.9773725

Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The emerging role of data scientists on software development teams. In *ICSE*. https://doi.org/10.1145/2884781.2884783

Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2017. Data scientists in software teams: State of the art and challenges. *TSE* (2017). https://doi.org/10.1145/3180155.3182515

H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre. 2011. HMDB: a large video database for human motion recognition. In *Proceedings of the International Conference on Computer Vision (ICCV)*. https://doi.org/10.1109/ICCV.2011.6126543

Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. 2020. The open images dataset v4. *International Journal of Computer Vision* (2020), 1–26. https://doi.org/10.48550/arXiv.1811.00982

Chris Lewis and Jim Whitehead. 2010. Runtime repair of software faults using event-driven monitoring. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 275–280. https://doi.org/10.1145/1810295.1810352

Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. 2017. Metamorphic model-based testing of autonomous systems. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. https://doi.org/10.1109/MET.2017.6

Tilo Linz. 2020. Testing Autonomous Systems. In *The Future of Software Quality Assurance*. Springer, Cham, 61–75.

Lisa-Assistant. 2021. A voice assistant application. Online document https://github.com/AlexNguyen27/lisa-assistant-gcp.

Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. *ACM SIGPLAN Notices* 49, 6 (2014), 227–238. https://doi.org/10.1145/2594291.2594337

David Marby and Nijiko Yonskai. 2021. Pyan3: Offline call graph generator for Python 3. Online document https://github.com/davidfraser/pyan.

Ronaldo Messina and Jerome Louradour. 2015. Segmentation-free handwritten Chinese text recognition with LSTM-RNN. In *2015 13th International conference on document analysis and recognition (icdar)*. IEEE, 171–175. https://doi.org/10.1109/ICDAR.2015.7333746

Microsoft. 2022a. Microsoft Azure Cognitive Services. Online document https://azure.microsoft.com/en-us/services/cognitive-services.

Microsoft. 2022b. Microsoft Azure Image Tagging. Online document https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/concept-tagging-images.

Hong Minhee. 2021. Python interface for Wikidata. Online document https://pypi.org/project/Wikidata/.

Chaithanya Kumar Mummadi, Robin Hutmacher, Kilian Rambach, Evgeny Levinkov, Thomas Brox, and Jan Hendrik Metzen. 2021. Test-time adaptation to distribution shift by confidence maximization and input transformation. *arXiv preprint arXiv:2106.14999* (2021). https://doi.org/10.48550/arxiv.2106.14999

Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. 2022. Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. *Organization* 1, 2 (2022), 3. https://doi.org/10.1145/3510003.3510209

Luca Pajola and Mauro Conti. 2021. Fall of Giants: How popular text-based MLaaS fall against a simple evasion attack. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 198–211. https://doi.org/10.1109/EuroSP51992.2021.00023

Karen Panetta, Landry Kezebou, Victor Oludare, James Intriligator, and Sos Agaian. 2021. Artificial Intelligence for Text-Based Vehicle Search, Recognition, and Continuous Localization in Traffic Videos. *AI* 2, 4 (2021), 684–704. https://doi.org/10.23919/FRUCT49677.2020.9211020

Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 87–102. https://doi.org/10.1145/1629575.1629585

Plant-Watcher. 2018. A plant management application. Online document https://github.com/siwasu17/plant-watcher.

ProjectSyn. 2020. A voice-activated light application. Online document https://github.com/mochiliu/projectsyn.

RoomR. 2020. A property management application. Online document https://github.com/rodrigoHM/RoomR-Server.

Amelie Royer and Christoph H Lampert. 2015. Classifier adaptation at prediction time. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1401–1409.

SeeFarBeyond. 2022. A coin finder application. Online document https://github.com/arosloff/SeeFarBeyond.

Apeksha Shewalkar. 2019. Performance evaluation of deep neural networks applied to speech recognition: RNN, LSTM and GRU. *Journal of Artificial Intelligence and Soft Computing Research* 9, 4 (2019), 235–245.

SmartCan. 2019. An garbage classification application. Online document https://github.com/ertheosiswadi/smart_can.

Kevin D. Smith, Jim J. Jewett, Skip Montanaro, and Anthony Baxter. 2022. PEP 318 âĂŞ Decorators for Functions and Methods. Online document https://peps.python.org/pep-0318/.

Victor Stinner. 2021. Bytecode: Python module to generate and modify bytecode. Online document https://pypi.org/project/bytecode/.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *ICLR*. https://doi.org/10.1109/CRV.2019.00010

Paul Voigtlaender and Bastian Leibe. 2017. Online adaptation of convolutional neural networks for video object segmentation. In *BMVC*. https://doi.org/10.48550/arXiv.1706.09364

Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2021. Are Machine Learning Cloud APIs Used Correctly?. In *43th International Conference on Software Engineering (ICSE'21)*. https://doi.org/10.1109/ICSE43902.2021.00024

Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. 2022. Automated Testing of Software that Uses Machine Learning APIs. In *44th International Conference on Software Engineering (ICSE'22)*. https://doi.org/10.1145/3510003.3510068

Chengcheng Wan, Yuhan Liu, Kuntai Du, Henry Hoffmann, Junchen Jiang, Michael Maire, and Shan Lu. 2023a. Artifact for "Run-Time Prevention of Software Integration Failures of Machine Learning APIs". Online document https://zenodo.org/record/8106520.

Chengcheng Wan, Yuhan Liu, Kuntai Du, Henry Hoffmann, Junchen Jiang, Michael Maire, and Shan Lu. 2023b. SmartGear. Online document https://github.com/mlapistudy/SmartGear.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. 2021b. Tent: Fully Test-time Adaptation by Entropy Minimization. In *ICLR*. https://doi.org/10.48550/arXiv.2006.10726

Deng-Bao Wang, Lei Feng, and Min-Ling Zhang. 2021a. Rethinking calibration of deep neural networks: Do not be afraid of overconfidence. *Advances in Neural Information Processing Systems* 34 (2021), 11809–11820.

Jonatas Wehrmann, Ricardo Cerri, and Rodrigo Barros. 2018. Hierarchical multi-label classification networks. In *International conference on machine learning*. PMLR, 5075–5084. https://doi.org/10.18653/v1/2022.emnlp-main.610

Wikidata. 2022. A free and open knowledge base. Online document https://www.wikidata.org/.

Erik Wittern, Annie TT Ying, Yunhui Zheng, Julian Dolby, and Jim A Laredo. 2017. Statically checking web API requests in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 244–254. https://doi.org/10.1109/ICSE.2017.30

Ruihan Wu, Chuan Guo, Awni Hannun, and Laurens van der Maaten. 2021. Fixes that fail: Self-defeating improvements in machine-learning systems. *Advances in Neural Information Processing Systems* 34 (2021), 11745–11756. https://doi.org/10.48550/arXiv.2103.11766

Shuzhao Xie, Yuan Xue, Yifei Zhu, and Zhi Wang. 2022. Cost Effective MLaaS Federation: A Combinatorial Reinforcement Learning Approach. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 1–10. https://doi.org/10.1109/INFOCOM48880.2022.9796701

Keiji Yanai and Yoshiyuki Kawano. 2015. Food image recognition using deep convolutional network with pre-training and fine-tuning. In *2015 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*. IEEE, 1–6. https://doi.org/10.1109/ICMEW.2015.7169816

Yuanshun Yao, Zhujun Xiao, Bolun Wang, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. 2017. Complexity vs. performance: empirical analysis of machine learning as a service. In *Proceedings of the 2017 Internet Measurement Conference*. 384–397.

Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *ASE*. IEEE, 132–142. https://doi.org/10.1145/3238147.3238187

Xinghan Zhao and Xiangfei Gao. 2018. An ai software test method based on scene deductive approach. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 14–20. https://doi.org/10.1007/978-3-031-11713-8_21

Luisa Zintgraf, Kyriacos Shiarli, Vitaly Kurin, Katja Hofmann, and Shimon Whiteson. 2019. Fast context adaptation via meta-learning. In *International Conference on Machine Learning*. PMLR, 7693–7702. https://doi.org/10.48550/arXiv.1810.03642

Received 2023-04-14; accepted 2023-08-27