Are Machine Learning Cloud APIs Used Correctly?

Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, Shan Lu
University of Chicago
{cwan, shicheng2000, hankhoffmann, mmaire, shanlu}@uchicago.edu

***Abstract*—Machine learning (ML) cloud APIs enable developers to easily incorporate learning solutions into software systems. Unfortunately, ML APIs are challenging to use correctly and efficiently, given their unique semantics, data requirements, and accuracy-performance tradeoffs. Much prior work has studied how to develop ML APIs or ML cloud services, but not how open-source applications are using ML APIs. In this paper, we manually studied 360 representative open-source applications that use Google or AWS cloud-based ML APIs, and found 70% of these applications contain API misuses in their latest versions that degrade functional, performance, or economical quality of the software. We have generalized 8 anti-patterns based on our manual study and developed automated checkers that identify hundreds of more applications that contain ML API misuses.**

I. Introduction

A. Motivation

Machine learning (ML) provides efficient solutions for a number of problems that were difficult to solve with traditional computing techniques; e.g., object detection and language translation. ML cloud APIs allow programmers to incorporate these learning solutions into software systems without designing and training the learning model themselves [1], and hence put these powerful techniques into the hands of non-experts. Indeed, there are more than 35,000 open-source projects on GitHub that use Google or Amazon ML Cloud APIs to solve a wide variety of problems, among which more than 14,000 were created within the last 12 months.

While these APIs make it easy for non-experts to incorporate learning into software systems, there are still a number of challenges that must be addressed to ensure that the resulting applications are both correct and efficient. While certain challenges come with the use of any third-party API, this paper focuses on unique challenges for ML APIs that arise due to the nature of learning itself.

Complicated data requirements. Machine learning techniques are used to process digitalized real-world visual, audio and text content. Although such content can be generated by a huge variety of devices and encoding software, the suitable input content and format (encoding, resolution, size, etc.) for ML APIs are rather limited and often uniquely defined by the DNN-training process. For example, cameras can produce images in many formats, but the image sets on which ML models are trained have a relatively small variety [2]–[8]. Thus, it is up to the API user to select the input or convert the input into what the API can accept and effectively process.

Complicated cognitive semantics. Unlike traditional APIs that are coded to perform well-defined algorithms, ML APIs are *trained* to perform cognitive tasks whose semantics cannot be reduced to concise mathematical or logical specifications, with inevitable overlap between different tasks; e.g., to detect a book in a scene, a user might call either `image-classification` or `object-detection`. Users need a good understanding of these cognitive semantics underlying ML APIs to pick the right API for the corresponding software component and usage scenario. Additionally, learning models operate in a continuous space (even if they ultimately produce a discrete output, the discretization is the last step in the model). Thus, it is up to users to understand the result of these calls and ensure that they know how to use the result correctly in the context of the software system.

Complicated tradeoffs. While many APIs offer tradeoffs between engineering effort and performance (e.g., higher performance APIs are more difficult to use), ML APIs have additional tradeoffs to consider. The first is accuracy. As ML APIs do not produce discrete "correct" or "incorrect" answers, it is up to users to understand the probabilistic nature of these API calls, how different data transformation and API selection can affect the accuracy, and the exact accuracy requirement of the corresponding software component. Furthermore, the engineering effort involved in using ML APIs is often related to transforming the input data, which can have large effects on performance and accuracy. Finally, as these APIs perform computation in the cloud, there is a monetary cost associated with every call, which is again affected by data transformation and API selection, and is yet another tradeoff to consider. It is essential that users understand the engineering/performance/accuracy tradeoffs of every ML API call and ensure that their application's requirements are met.

If ML API users do not address the above challenges, their software systems can suffer from inefficiencies (in performance or cost) and correctness issues. In addition, the fact that these APIs do not produce binary correct/incorrect outputs means that the resulting performance and accuracy losses can be difficult to diagnose; e.g., in addition to catastrophic fail-stop failures (which are at least easy to notice), misunderstanding the API semantics produces lower accuracy and higher cost software. Thus, while these APIs make it possible for non-expert users to incorporate ML into software systems, it is still necessary that users understand and avoid API misuses.

Prior work studies software development for ML. For example, recent work proposes methods for finding bugs in ML libraries [9]–[15]. Other work finds bugs related to designing and training ML models [16]–[49]. However, to the best of our

knowledge no prior work provides an empirical study detailing the software engineering issues that arise when calling third-party ML APIs from within software systems.

B. Contributions

To understand the problems that arise when using ML cloud APIs and design appropriate solutions, we perform an empirical study of the *latest* versions—as of August 1, 2020—of 360 GitHub projects that include non-trivial use of Google Cloud and Amazon AWS APIs, the two most popular AI services, and cover all the three ML domains offered by them: vision, speech, and language.

Our study faces the challenge of lacking existing issue-tracking system records about ML API misuses, given the short history of ML APIs. Consequently, we carefully study these 360 projects and discover previously *unknown* misuses in their latest versions by ourselves.

Our study found that misuses of ML APIs are widespread and severe: 247 out of these 360 applications (69 %) contain misuses in their latest versions, more than half of which contain more than one type of misuse.

These misuses lead to various types of problems, including 1) *reduced functionality*, such as a crash or a quality-reduced output; or 2) *degraded performance*, like an unnecessarily extended interaction latency; or 3) *increased cost*, in terms of payment for cloud services. Their root causes are all related to unique challenges for ML APIs discussed above, which we present in detail in Sections IV, V, and VI.

Our study reveals common misuse patterns that are found in many different applications, often with simple fixes that avoid failures, improve performance, and reduce cost. Therefore, as a final contribution, we design several checkers and small API changes (in the form of wrapper functions) that both check for and handle common errors. Many more misuses are found by our checkers, beyond the 360 projects in the initial study. We present solutions to some of the problems we have uncovered in Section VII.

Overall, this paper presents the first in-depth study of real-world applications using machine learning cloud APIs. It provides guidance to help prevent errors while improving the functionality, performance, and cost of these applications.

We have released our whole benchmark suite, automated checkers, and detailed study results online [50].

II. Background

Several companies provide a broad set of machine learning cloud services, such as Google Cloud AI [52], Amazon Web Service (AWS) AI [53], IBM Watson [54], and Microsoft Azure [55]. These services are built upon pre-trained DNNs designed to tackle specific problems. They each offer a set of APIs. By calling these APIs, inference computations that use industry-trained DNNs can be conducted on powerful cloud servers without requiring developers to understand details about machine learning or conduct resource provision.

As shown in Table I, these cloud services cover three ML domains. (1) **Vision**. This includes image-oriented and video-oriented machine-learning tasks, like detecting objects, faces, landmarks, logos, text, or sensitive content from an image or a video. (2) **Language**. This includes natural language processing (NLP) tasks, like detecting or analyzing entity, sentiment, language, or syntax from text inputs. It also includes translation tasks. (3) **Speech**. This includes recognizing text from an audio input, and synthesizing an audio from text input.

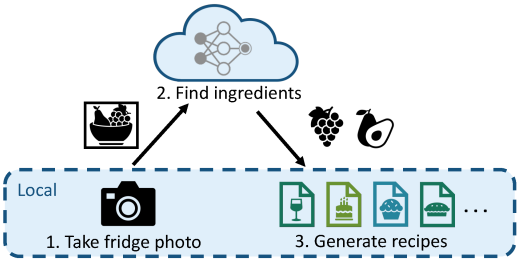


Fig. 1: An example of using ML APIs [51].

Figure 1 illustrates an example of how applications use ML APIs. It depicts the workflow of **Whats-In-Your-Fridge** [51], an open-source GitHub application for recipe suggestion. This application uploads a photo taken inside the fridge to the cloud, applies a vision API to find out what is inside the fridge, and then generates recipes accordingly. Of course, as we will discuss later, this application actually cannot deliver its functionality due to an API misuse.

III. Methodology

A. Application selection

Our work looks at applications that use Google Cloud AI and Amazon AI, the two most popular cloud AI services on Github, with thousands of applications using each type of their AI services, as shown in Table II. Our work will target the following two sets of applications (all latest versions as of *Aug. 1st, 2020*), one for all our manual studies and one for our automated checking.

For automated checking, we use *all* the 12666 Python applications on GitHub that use Google or AWS AI service.

For manual studies, we collect a suite of 360 non-trivial applications that use Google/Amazon ML APIs, including 120 applications for each of the three major ML domains. They cover different programming languages, Python(80%), JS (13%), Java (3%), and others (4%). Around 80% of these applications use Google Cloud AI and around 20% use AWS AI, with 1% using both. We used fewer applications that use AWS AI service, as AWS Lambda [56], a serverless computing platform, sometimes makes it difficult for us to judge the exact application workflow. The sizes of these applications range from 46 to 3 millions lines of code, with 2228 lines of code being the median size and around 40% of them having more than 10 thousand lines of code. Most of these applications are young, created after 2018 (98% of them). They have a median age of around 18 months at the time of our study. This relatively young age distribution reflects the fact that the power of deep learning has only been recently recognized, and yet is being adopted with unprecedented pace and breadth.

		Google Cloud AI	AWS AI	IBM Cloud Watson	Microsoft Azure Cognitive Services
Vision	Image	Vision AI	Rekognition	Visual Recognition$_S$	Computer Vision, Face
	Video	Video AI		-	Video Indexer$_A$
Language	NLP	Cloud Natural Languages$_S$	Comprehend	Natural Language Understandings$_S$	Text Analytics
	Translation	Cloud Translation$_S$	Translate$_S$	Language Translator	Translator
Speech	Recognition	Speech-to-Text	Transcribe$_A$	Speech to Text	Speech to Text
	Synthesis	Text-to-Speech$_S$	Polly	Text to Speech$_S$	Text to Speech

TABLE I: ML tasks supported by four popular ML cloud services. Subscript $_S$: only a synchronous API is offered for this task; subscript $_A$: only an asynchronous API is offered; no subscript: both synchronous and asynchronous APIs are offered.

		All Apps		New Apps	
		Google	AWS	Google	AWS
Vision	Image	7916	8818	4221	2951
	Video	674		231	
Language	NLP	4632	4291	2341	1969
	Translation	1192	7681	476	2865
Speech	Recognition	9439	5155	3291	2222
	Synthesis	2190	6375	1037	1986
Total (w/o duplicates)		35376		14049	

TABLE II: # of applications using different types of ML APIs on GitHub. New Apps refer to those created after 08-01-2019.

Since there are many toy applications on GitHub, we manually checked about 1200 randomly selected applications, which use Google/Amazon ML APIs, to obtain these 360 non-trivial applications. We manually confirmed they each target a concrete real-world problem, integrate the ML API(s) in their workflow, and conduct some processing for the input or the output of the ML API, instead of simply feeding an external file into the ML API and directly printing out the API result. We do not have a way to accurately check how seriously these applications have been used in the real world, and it is possible that some of these 360 applications have not been widely used.

B. Anti-pattern identification methodology

Because of the young ages of ML API services and hence the applications under study, we could *not* rely on known API misuses in their issue-tracking systems, which are very rare. Instead, we must discover API misuses *unknown to the developers* by ourselves.

Since there is no prior study on ML API misuses, our misuse discovery can not rely on any existing list of anti-patterns. Instead, our team, including ML experts, carefully studies API manuals, intensively profiles the API functionality and performance, and then manually examines every use of an ML API in each of the 360 applications for potential misuses. For every suspected misuse, we design test cases and run the corresponding application or application component to see if the misuse truly leads to reduced functionality, degraded performance, or increased cost comparing with an alternative way of using ML APIs, which we designed. When one misuse is identified, we generalize it and check if there are similar misuses in other applications. We repeat this process for many rounds until we converge to the results presented in this paper. During this process, we report representative misuses to corresponding application developers, receiving confirmation for many cases. All the manual checking is conducted by two of the authors, with their results discussed and checked by all the co-authors.

We identify a wide variety of applications as containing ML API misuses including those both: small and large, young and old, AWS and Google-API based. This variety of misuses indicates that they are not rare mistakes by individual programmers and do not appear to diminish with software growth, age, or API provider.

C. Profiling methodology

In section V, we profile several projects to evaluate their performance before and after optimization. We use real-world vision, audio, or text data that fits the scenario of corresponding software. We profile the end-to-end latency for each related module and also the whole process: from user input to final output. By default, we run each application under profiling five times for each input and reported the average latency.

All experiments were done on the same machine, which contains a 16-core Intel Xeon E5-2667 v4 CPU (3.20GHz), 25MB L3 Cache, 64GB RAM, and 6×512GB SSD (RAID 5). It has a 1000Mbps network connection, with twisted pair port. Note that all the machine-learning inference is done by cloud APIs remotely, instead of on the machine locally.

IV. Functionality-related API Misuses

Through manual checking, we identified three main types of API misuses that commonly affect the functional correctness of applications, as listed in Table III (white-background rows). They are typically caused by developers' misunderstanding of the semantics or the input data requirements of machine learning APIs, and can lead to unexpected loss of accuracy and hence software misbehavior that is difficult to diagnose.

Note that, although the high-level patterns of these misuses, such as calling the wrong API and misinterpreting the outputs, naturally occur in general APIs, the exact root causes, code anti-patterns, and tackling/fixing strategies are all unique to ML APIs, as we discuss below.

A. Calling the wrong API

Unlike traditional APIs that are *programmed* to each conduct a clearly coded task, ML APIs are *trained* to perform tasks emulating human behaviors, with functional overlap among some of them. Without a good understanding of these APIs, developers may call the wrong API, which could lead to severely degraded prediction accuracy or even a completely wrong prediction result and software failures. We discuss three pairs of APIs that are often misused below.

What challenges did developers encounter?	Related APIs and Inputs	Service Provider	Impact	# (%) of Problematic Apps. Manual	Auto
Should Have Called a Different API					
Complicated cognitive semantic overlap across APIs	text-detection vs. document-text-detection	G	Low Accuracy	6 (11%)	-
	image-classification vs. object-detection	AG	Low Accuracy	5 (9%)	-
	sentiment-detection vs. entity-sentiment-detection	G	Low Accuracy	4 (5%)	-
Complicated tradeoffs: Input-Accuracy-Perf.	ASync vs. Sync Language-NLP	A	Slower	-	3 (43%)
	ASync vs. Sync Speech Recognition	G	Slower	7 (78%)	203 (83%)
	ASync vs. Sync Speech Synthesis	A	Slower	-	2 (22%)
Unaware of parallelism APIs	Vision-Image API vs. annotate-image	AG	Slower	7 (78%)	-
	Language-NLP API vs. annotate-text	AG	Slower	11 (100%)	-
	Regular API vs Batch API	AG	Slower	Workload dependent	
Should Have Skipped the API call					
Complicated tradeoffs: Input-Performance	Speech Synthesis APIs with constant inputs	AG	Slower, More Cost	15 (25%)	279 (17%)
Complicated tradeoffs: Accuracy-Performance	Vision-Image APIs with high call frequency	AG	Slower, More Cost	3 (3%)	-
Should Have Converted the Input Format					
Complicated data requirements	all APIs without input validation, transformation	AG	Exceptions	206 (57%)	-
Complicated tradeoffs: Input-Accuracy-Perf.	Vision-Image APIs with high resolution inputs	AG	Slower	106 (88%)	-
Complicated tradeoffs: Input-Accuracy-Cost	Language-NLP APIs with short text inputs	AG	More Cost	4 (3%)	-
	Speech recognition APIs with short audio inputs	AG	More Cost	1 (2%)	-
	Speech synthesis APIs with short audio inputs	AG	More Cost	1 (2%)	-
Should Have Used the Output in Another Way					
Complicated semantics about outputs	sentiment-detection	G	Low Accuracy	24 (39%)	360 (37%)
Total number of benchmark applications with at least one API misuse		AG		249 (69%)	

TABLE III: ML API misuses identified by our Manual checking and Automated checkers. "A" is for AWS and "G" for Google. The %s of problematic apps are based on the total # of apps using corresponding APIs in respective benchmark suite. Note that, 133 apps contain more than one type of API misuses; the average number of API misuses in each application is 1.3.

`Text-detection` and `document-text-detection` are both vision APIs designed to extract text from images, with the former trained for extracting short text and the latter for long articles. Mixing these two APIs up will lead to huge accuracy loss. Our experiments using the IAM-OnDB dataset [57] show that `text-detection` has about 18% error rate in extracting hand-written paragraphs, and can only extract individual sentences—not complete paragraphs—when processing multi-column PDF files; yet, `document-text-detection` makes almost no mistakes for these long-text workloads. This huge accuracy difference unfortunately is not clearly explained in the API documentation and is understandably not known by many developers.

In our benchmark suite, 52 applications used at least one of these two APIs, among which 6 applications (11%) use the wrong API. For example, **PDF-to-text** [58] uses `text-detection` to process document scans, which is clearly the wrong choice and makes the software almost unusable for scans with multiple columns.

`Image-classification` and `object-detection` are both vision APIs that offer description tag(s) for the input image. The former offers one tag for the whole image, while the latter outputs one tag for every object in the image. Incorrectly using `image-classification` in place of `object-detection` can cause the software to miss important objects and misbehave; an incorrect use along the other direction could produce a wrong image tag.

In our benchmark suite, 57 applications use at least one of these two APIs, among which 5 applications (9%) pick the wrong API to use. For example, **Whats-In-Your-Fridge** [51] is expected to leverage the in-fridge camera to tell a user what products are currently inside the fridge. However, since it incorrectly applies `image-classification`, instead of `object-detection`, to in-fridge photos, it is doomed to miss most items in the fridge—a severe bug that makes this software unusable. Similarly, **Phoenix** [59] is expected to detect fire in photos and warn users, but incorrectly uses `image-classification`. Therefore, it is very likely to miss flames occupying a small area. We have reported this misuse to developers and they have confirmed this bug.

Similar problems also exist in language APIs. For example, `sentiment-detection` and `entity-sentiment-detection` can both detect emotions from an input article. However, the former judges the overall emotion of the whole article, while the latter infers the emotion towards every entity in the input article. Mis-use between these two APIs can lead to not only inaccurate but sometimes completely opposite results, severely hurting the user experience. In our benchmark suite, 86 applications used these APIs, among which 4 applications (5%) use the wrong one.

Summary Above API mis-uses form an important and new type of semantic bugs: the machine-learning component of software suffers unnecessary accuracy losses due to simple API-use mistakes, which we refer to as accuracy bugs. Accuracy bugs in general are difficult to debug, as they are difficult to manifest under traditional testing and developers may easily blame the underlying DNN design without realizing their own, easily fixable, mistakes. The particular accuracy bugs discussed here involve some of the most popular APIs, used by more than half of the applications in our suite, and hence are particularly dangerous. We reported some of these bugs to a few actively maintained applications recently, and already got two bug reports confirmed by developers.

One may tackle this problem through a combination of

```
response = client.analyze_sentiment(document=document,
                                    encoding_type=encoding_type)
...
sentiment = response.document_sentiment.score
...
if avg_sentiment < 0:
   message = '''Your posts show that you might not be '
                        going through the best of time. '''
```

Fig. 2: Misinterpreting outputs in **JournalBot** [62]

program analysis, testing, and DNN design support. Some of these misuses may be statically detected by checking how the API results are used—if only one tag or sentiment result is used following a `object-detection` or `entity-sentiment-detection` call, there is a likely mis-use. Mutation testing that targets these misuse patterns could also help—we can check whether the software behaves better when replacing one API with the other. Finally, it is also conceivable to extend the DNN or add a simple input classifier to check if the input differs too much from the training inputs of the underlying DNN, similar to the problem of identifying out-of-distribution samples tackled by recent ML work [60].

B. Misinterpreting outputs

Related to the probabilistic nature of cognitive tasks, DNN models operate on high-dimensional continuous representations, yet often ultimately produce a small discrete set of outputs. Consequently, ML APIs' outputs can contain complicated, easily misinterpretable semantics, leading to bugs.

A particularly common mistake concerns the sentiment detection API from Google's NLP service. This API returns two floating point numbers, `score` and `magnitude`. Among them, `score` ranges from -1 to 1 and indicates whether the input text's overall emotion is positive or negative; `magnitude` ranges from 0 to $+\infty$ and indicates how strong the emotion is. According to Google's documentation [61], these two numbers should be used together to judge the sentiment of the input text: when the absolute value of either of them is small (e.g., `Score` < 0.15), the sentiment should be considered neutral; otherwise, the sentiment is positive when `score` is positive and negative when `score` is negative. In our benchmark suite, 62 applications have used this API, among which 24 have used the API results incorrectly (39%).

For example, a journal app **JournalBot** [62] (Figure 2) uses this API to judge the emotion in a user's journal and displays encouraging messages when the emotion is negative. Unfortunately, it considers the journal to be emotionally negative checking only that `score` < 0. This interpretation often leads to wrong results and hence unfitting messages—when the `magnitude` is small or the `score` is a small negative value, the emotion should be neutral even if `score` < 0. We have reported it to developers and they confirmed this bug.

Summary Incorrectly using ML API results can again lead to accuracy bugs that are difficult to debug. We reported some of these bugs to a few actively maintained applications recently, and already got three bugs confirmed by developers.

This above problem about sentiment detection can be alleviated by automatically detecting result misuse through static program analysis, which we discuss in Section VII.

C. Missing input validation

Inputs to ML APIs are typically real-world audio, image, or video content. These inputs can take many different forms, with different resolutions, encoding schemes, and lengths. Unfortunately, developers sometimes do not realize that not all forms are accepted by ML APIs, nor do they realize that such input incompatibility can be easily solved through format conversion, input down-sampling, or chunking. As a result, lack of input validation and incompatibility handling are very common, and can easily cause software crashes.

Many ML APIs have input requirements and an exception is thrown at an incompatible input. For example, the Google speech recognition APIs have formatting requirements (i.e., single channel, using 16 bit samples for LINEAR_PCM) and size requirements ($<$ 1 minute for synchronous APIs) for audio inputs; vision APIs have size requirements (i.e., $<$ 5 MB for AWS and $<$ 10 MB for Google) for image inputs.

Among the 360 benchmark applications, 11% choose to use APIs that do not require input validation, about one third make the effort to guarantee their input validity through input checking and transformation, and yet more than half of the applications made no effort to guarantee input compatibility (206 applications). Furthermore, none of these 206 applications handle exceptions thrown by API calls, and hence can easily encounter software crashes due to incompatible inputs.

For example, **Automatic-Door** [63] takes input camera images and decides to open or close a door using face verification through the AWS API `compare-faces`. Since `compare-faces` requires the input image to be smaller than 5 MB, without any input checking and transformation, this software could be completely unusable if it happens to be deployed with a high resolution camera.

Summary Input checking and transformation is particularly important for ML APIs, considering the wide variety of real-world audio and visual content, and is unfortunately ignored by developers at an alarming rate—206 out of 360 applications, severely threatening software robustness. This problem can be alleviated by automatically detecting and warning developers of the lack of input validation or exception handling. Even better, we can design a wrapper API that automatically conducts input checking and transformation (e.g., image down-sampling and audio chunking), which we will present in Section VII.

V. Performance-related API Misuses

Through manual checking, we identify and categorize 4 main types of ML API mis-uses that can lead to huge performance loss and user experience damage (see Table III, blue-background rows). They are typically related to ML APIs' complicated tradeoffs among input-transformation effort, performance, and accuracy.

A. How important are performance anti-patterns?

To motivate the study below, we first check whether the performance of ML APIs matters for software user experience.

First, the latency of ML APIs are significant, ranging from close to one second to several minutes for typical inputs. Based on our profiling, in vision tasks, most APIs takes 0.2-0.6 seconds to process a low-resolution image with 550×400 pixels, and almost one full second to process a high-resolution image. In language tasks, a 5000-character input takes 0.60 ($\pm$ 0.05) seconds for synchronous APIs and as many as 413 ($\pm$ 58) seconds for asynchronous APIs.[1] In speech tasks, a 30-second short audio clip takes 7.1 ($\pm$ 1.5) seconds with synchronous APIs and 13.6 ($\pm$ 4.9) seconds with asynchronous APIs.[2]

Second, we find that more than one third of the benchmark applications have (soft) latency deadlines of a couple of seconds or less, with their service quality directly affected by ML APIs. Many of them (114 out of 360) involve ML APIs in their critical user-interactive workflow and hence need the API result to return within a couple of seconds to maintain good software interactivity [64], [65]; in addition, some applications (11 out of 360) process streaming data, audio, video, and others, from a sensor, and hence have to finish each API call in less than one second [66] to avoid data loss. Even for those applications that do not have tight deadlines, typically one would still hope an output to be generated in a few minutes, which could still be challenging, as these applications typically feed a large amount of data to ML APIs.

Clearly, inefficient use of ML APIs can cause severe damage to user experience, as we will see in real examples below.

B. Misuse of asynchronous APIs

The same ML task can often be performed with multiple APIs, a synchronous version, an asynchronous version, and sometimes a streaming version (see Table I). The different versions have complicated and sometimes counter-intuitive tradeoffs between input transformation, performance, and accuracy that often confuse developers and lead to surprisingly wide-spread and severe misuses based on our study.

A common problem is related to asynchronous ML APIs. In many concurrent programs, asynchronous functions are used to gain performance through improved concurrency at the cost of extra development effort. In most ML applications, the tradeoff is the *opposite*: asynchronous ML APIs are called without improved concurrency and huge performance loss in exchange for less effort in input transformation.

The benefit of asynchronous ML APIs is clearly documented: they allow much longer audio/text inputs than synchronous APIs. For example, in Google speech recognition service, the synchronous API takes audio up to 1-minute long, while the asynchronous API can take up to 480 minutes [67].

The performance downside of asynchronous APIs is unfortunately *not* quantitatively specified in the documentation. In our profiling, synchronous and streaming APIs are about twice as fast as asynchronous APIs in Google Speech-to-Text service, as shown in Figure 3.a. The difference is even bigger for AWS Comprehend service (i.e., NLP). Since its multi-file synchronous API has built in parallelism, the speed up over asynchronous API can be as many as 400X (Figure 3.b).

Making things worse, most applications call asynchronous ML APIs synchronously, with the caller blocking itself until the API returns and no other concurrent execution on going, and hence has no way to compensate for the poor performance. Among the 44 benchmark applications using Google speech recognition APIs, 9 use the asynchronous API. 7 out of 9 make the asynchronous call in a synchronous way. Our automated checker confirms this trend: 203 out of 246 GitHub applications call this asynchronous API in a synchronous way.

Clearly, many of these asynchronous APIs could be replaced with synchronous or streaming APIs, with a huge performance improvement (up to 400X as profiling shows). We demonstrate these optimizations using a few benchmark examples below.

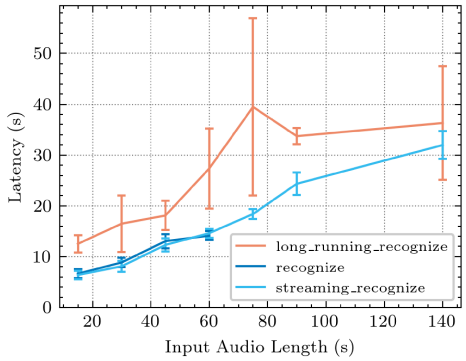
Replacing with synchronous call. Answering-Machine [68] applies the asynchronous speech recognition API to every voice mail and then sends specific text messages to slack accounts based on the transcript returned by the API call. Since the typical length of a voice mail is 30 seconds [69], it could have checked the size of every voice mail first, which takes 0.002 seconds in our profiling, and then used the synchronous API for most of the voice mails with a huge speedup: for a 30.0-second voice mail, the asynchronous Speech-to-Text API takes 16.5 ($\pm$ 5.9) seconds and yet the synchronous API takes only 8.9 ($\pm$ 1.0) seconds—a huge latency improvement.

Jiang-Jung-Dian [70] is an application that automatically generates meeting reports. It needs more than 8 minutes (i.e., 490 seconds) to process a one-minute meeting recording (all numbers are averaged based on five runs). Our profiling shows that uploading the audio file and downloading the results together only take 1.1 seconds, and yet the majority of the time is spent in an asynchronous Speech-to-Text API call and then an asynchronous text Comprehend API call, with the latter alone taking close to 7 minutes (410 seconds). If we replace it with AWS synchronous multiple-file Comprehend API, the API execution time drops from 410 seconds down to only 0.97 seconds (more than 400X speedup!), and hence is no longer a performance bottleneck. In fact, the AWS synchronous multi-file Comprehend API can take in 25 documents at a time with each document containing up to 5000 characters, big enough to hold the transcript of several hours' meetings.

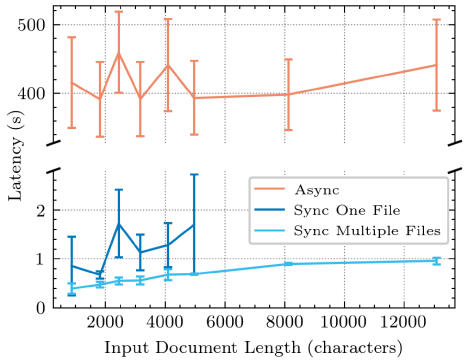
Replacing with streaming call. Much real-world audio content takes a streaming form, and is supported by streaming APIs for several audio-related ML tasks, like the speech recognition service in Google Cloud [71] (AWS offers streaming APIs but not for Python programs). These streaming APIs can either be directly applied to a local audio file, which was the setting in Figure 3.a, or to a streaming input. They offer unique benefits for a streaming input: (1) they can start processing input and returning inference results before the whole audio finishes; (2) they support an unlimited length of

[1]Profiled with AWS Comprehend on three types of inputs: a philosophy text, a novel with conversations, and a CNN news article.

[2]Profiled with Google Speech-to-Text on three different inputs: a news broadcast, an online lecture, and a WSJ audio. Data format: avg ($\pm$ std)



(a) Google Speech-to-Text



(b) AWS Comprehend

Fig. 3: Latency profiling for three different APIs of Google Speech-to-Text (synchronous, asynchronous, and streaming) and AWS Comprehend (synchronous one file, synchronous multi-file, and asynchronous). Each point in the figure corresponds to the mean and the error bar corresponds to the standard deviation of five experiments. Note that, in (b) the y-axis is broken into two parts with different value ranges.

stream input, so that we do not need to worry about chunking a large file or having to make a slow asynchronous call. Unfortunately, developers sometimes call non-streaming APIs to process streaming input, causing much performance loss.

In our benchmark suite, 29 applications use Google synchronous or asynchronous speech recognition APIs. Among them, 4 applications are actually working on streaming inputs, and can be greatly optimized by switching to the corresponding streaming API `streaming_recognize`.

For example, **Potty-Pot** [72] detects offensive language in audio streamed from a microphone. It repeatedly records the microphone input in a 5-second audio clip, feeds it into a synchronous Google speech recognition API to look for spotted words, which takes around 5 seconds, and then records the next 5-second audio clip, and so on. This can lead to severe quality of service problems: either a big portion of the microphone audio will not be checked or the users have to carefully pace their speaking, pausing 5 seconds after every 5 seconds of speaking. Instead, with a streaming API, the user experience will be much improved: based on our profiling, after speaking for 5 seconds, the user only needs to wait for an extra 0.1 second for all the checking to finish.

As another example, **Class-Scribe-LE** [73] records lecture audio and then calls the asynchronous API to generate lecture notes. As a result, after a two minute lecture audio is played, one needs to wait for almost 3 minutes for the notes to generate and yet only 2 seconds if a streaming API is used, which thus accomplishes most of the work during the lecture time.

Summary: The complicated tradeoff among synchronous, asynchronous, and streaming APIs has clearly confused many developers. This leads to a broad misuse of asynchronous APIs, as quantified in Table III, and severe performance loss and user experience damage. We could create a wrapper API that makes the choice for developers (Section VII).

C. Forgetting parallel APIs

Some ML APIs are offered to ease task and data parallelism, but are rarely used even when doing so would require only a simple change to the application.

Forgetting task parallelism. Both Google and AWS offer task-parallelism through easy-to-use APIs, `annotate-image` and `annotate-text`. Multiple vision or NLP services can be specified as parameters of these two APIs, and then each service is applied to the same input in parallel.

Unfortunately, among the 20 benchmark applications that apply multiple vision (NLP) APIs towards the same input image (text), only 2 of them use the `annotate-image` (`annotate-text`) API. The majority of them completely miss this easy parallelism opportunity. For example, Okuninushi [74], a website for Japanese wine database, applies ImageClassification and TextDetection to every input image sequentially. An easy refactoring to use `annotate-image` offers 2X speedup. We have reported this problem to developers and they have confirmed this bug.

Forgetting data parallelism. Google and AWS both offer data-parallelism through easy-to-use batching APIs, which take multiple input files and process them at once. This offers optimization opportunities for those applications with large inputs: the large input can be chunked into multiple smaller pieces and get processed using a batching API.

Of course, this optimization depends on the specific workload and task. First, the workload should be large enough to amortize the extra input and output processing cost. Second, the ML task needs to make sure that the aggregated results from input chunks are (mostly) the same as the original result from processing one big file. This works for speech synthesis, speech recognition, entity detection, and syntax analysis tasks, as long as the input audio or text is carefully chunked, like at the boundaries of pauses, sentences, or paragraphs.

For example, **EmailClassifier** [75] downloads all the emails saved in a database and then applies the AWS NLP API to detect sentiments and extract entities from every email. We can easily chunk long emails by paragraph and then process all paragraphs in parallel using the batching API. Particularly, chunking by paragraph typically has no effect to the accuracy of keyword extraction and entity recognition tasks [76], [77]. The results produced by the synchronous one file API and the

synchronous multiple files API only have very minor word difference, with the latter offering a 1.5X speedup for a 4500-character sample email (0.44 seconds vs. 0.66 seconds). The total time saving for all the emails will be significant.

Samaritan [78] is another example. It first uses a speech recognition API to get transcript from a doctor's voice message, and then uses an NLP API to detect entities from the transcript. In addition to the entity-detection task discussed above, the speech recognition task is also suitable for a batching optimization: chunking an audio file by silence every 10-15 seconds typically has minor impact on the output, as speech recognition DNNs usually are trained on short audio snippets (e.g. VCTK dataset [79] mostly consists of 2-6 second audio clips, and Google Audioset [80] consists of less than 10 second audio clips). Furthermore, a doctor's voice message is often long enough to get chunked into multiple 10–15 second clips which can be processed in parallel.

Summary: The mentioned parallelism APIs are rarely used in our benchmark suite, appearing in only 1 out of the 360 applications. Static analysis can be used to identify ML APIs sequentially applied to the same input data, and suggest or automate an optimization that uses `annotate*` APIs. By dynamically checking the input size to some ML APIs like speech recognition and entity detection, data-parallel optimization can be done by calling batch APIs, which we have implemented as API wrappers (Section VII).

D. Making skippable API calls

Sometimes, an API call can be skipped at the cost of slightly higher engineering effort or slight, but often indiscernible by human, functionality difference. Lack of understanding of these tradeoffs leads to some unnecessary API calls.

API calls with constant inputs. Among the 60 benchmark applications that use the speech synthesis API, 15 (25%) of them call this API with a constant string input and thus could have replaced the API call with a pre-recorded audio. As we will see in Section VII, our automated checker found that this is indeed a prevalent problem in hundreds of applications.

An example is **Sounds-Of-Runeterra** [81] (Figure 4), a card game extension that improves game accessibility to visually impaired users. It contains multiple unnecessary calls to Google speech synthesis API, each generating an audio clip for one constant string, e.g., "You won", "Exiting application", etc. Replacing each of them with a pre-recorded audio clip can save 0.9 seconds and associated monetary cost for each API call.

API calls with excessive frequency. Sometimes, a program repeatedly invokes an image-processing API at high frequency. Reducing the invocation frequency can lead to huge performance improvement with little to no perceivable output difference to human users. Among 120 vision benchmarks, 3 of them fall into this anti-pattern.

For example, **Ns-Tool** [82] is a game screen monitoring application. Every second, it takes a screenshot of the game and applies the `text-detection` API to check whether the screen is locked; if so, it sends a message through the

```
def _stop(self):
  audio = self.transform_text_to_audio_as_bytes_io(
                                      "Exiting application.")
  ...
def transform_text_to_audio_as_bytes_io(self, string,
                        language_code = DEFAULT_LANGUAGE_CODE):
  voice_request = build_voice_request(string, language_code)
  response = self.client.synthesize_speech(
                                voice_request.synthesis_input,
                                  voice_request.voice_config,
                                  voice_request.audio_config)
  ...
```

Fig. 4: Skippable call@ **Sounds-Of-Runeterra** [81]

internet to the user. Clearly, this causes unnecessary waste of computation resources, because the auto-sleep duration is at least several minutes and a couple of seconds' delay in sending out the reminder message would not matter to users. As another example, **Tags** [83] is a video scene-detection application. It applies the image classification API to analyze every frame of the input video; it then splits the video into smaller pieces based on where the image-classification output changes; and eventually outputs the video splits and the label of each split to the user. Clearly, we could apply the image classification API at a much sparser rate (e.g., once every other frame or even sparser) with big performance improvement and little impact to output quality, as most of the adjacent video frames are similar to each other and a miss of a couple of frames is probably un-perceivable to human eyes.

Summary: These problems also occur with other APIs as well, although not as common as that for speech synthesis and vision-image APIs. We reported some of the constant-input speech-synthesis problems to a few actively maintained applications recently, and already got three bugs confirmed.

We have built a static checker to automatically identify speech synthesis API call with a constant input (Section VII); future research could design a dynamic controller to adjust API call frequency, balancing functionality and performance.

E. Unnecessarily high-resolution inputs

Vision APIs accept inputs with a range of resolutions and impose a complicated tradeoff among input, performance, and accuracy that is often ignored by developers—with higher input resolution, the performance degrades greatly, while the inference accuracy increases and then saturates quickly.

This tradeoff is not explained clearly in the tutorial: AWS tutorial did not offer any resolution suggestion; Google vision APIs did suggest image resolution to be 640 x 480, which is ignored by most developers. To better understand this tradeoff, we conducted an experiment with 100 randomly collected high resolution images in four categories (Dog, Bufferfly, Scooper, and Wardrobe). We down-sampled each image to create 6 more images with different resolutions as shown in Figure 5, and then feed them each into the Google image classification API. As shown in the figure, the round-trip API

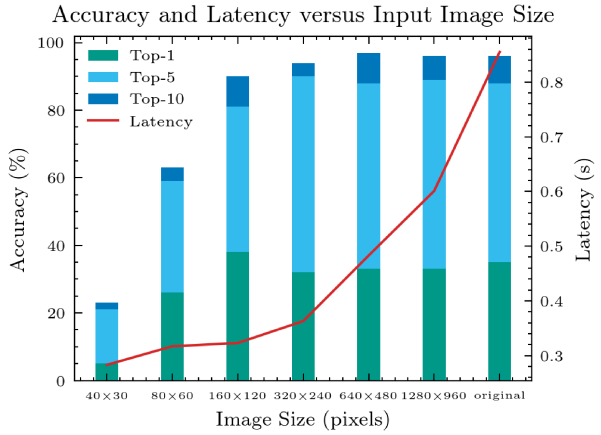


Fig. 5: Accuracy and latency with different input resolutions.

time increases greatly with resolution and yet the accuracy saturates at 640 x 480. A likely reason is that most vision datasets [2]–[4], [6]–[8], on which vision DNNs are trained, contain images with similar resolutions ranging from 32 x 32 to 1100 x 700. Consequently, higher resolutions do not lead to higher accuracy. Note that, down-sampling an image takes only 0.03 seconds on average, negligible comparing with the API latency. Due to space constraints, we omit the AWS results here, which have a similar trend.

Given this tradeoff, developers really should follow the tutorial suggestion in feeding relatively low resolution images (e.g., 640 x 480) into vision APIs. However, among the 120 applications in our benchmark suite that use Vision-Image APIs, only 9 of them stick to this guideline by down-sampling every high-resolution user input. The remaining 106 applications all waste performance without accuracy benefit for any input that has higher than 640 x 480 resolution, which unfortunately is the majority today.

Summary: Without a clear understanding about the accuracy-performance tradeoff, most developers ignore input transformation (down-sampling). A static checker could issue warnings for lack of input transformation. A run time controller can also decide the ideal input size based on applications' accuracy and performance constraints.

VI. Cost-related API Misuses

Every ML API call costs money. Naturally, some performance problems, particularly all of those skippable calls in Section V-D, also waste money. In addition, the round-up charging policy—shown in Table IV—leads to a unique anti-pattern: since every API call is charged based on the input size rounded-up, calls with very small inputs may be economically sub-optimal. The possibility of combining multiple calls with small inputs creates a complicated tradeoff problem among input transformation, accuracy, performance, and cost.

Without knowing the exact input distribution, it is difficult to identify applications that fall into this anti-pattern. Nevertheless, our benchmark suite contains some examples.

		Pricing Unit	Price ($)
Vision	Image	1 image	1.5-3.5 per 1000 unit
	Video	1 minute	0.05-0.15 per unit
Language	NLP	1000 characters	0.25-1 per 1000 unit
	Translation	1 character	20 per million character
Speech	Recognition	15 seconds	6-9 per 1000 unit
	Synthesis	1 character	4-16 per million character

TABLE IV: Cost of Google cloud AI services.

Audio-Sentence-Split [84] takes any input audio, slices it into 1- to 2- second audio clips based on silence in the audio, feeds the clips one by one to the Google speech recognition API, and finally stores the resulting pairs of clip–transcript into a database. Since every API call is charged based on the audio length rounded up to multiple of 15 seconds, chunking into 1- or 2- second snippets wastes money and likely hurts inference accuracy, as well. A more cost-efficient implementation is to feed the whole audio into one API call and then slice the returned transcript and audio, in whatever way the application sees fit (the returned transcript contains information about the exact audio position matched to each word, which makes chunking easy). For example, a 60-second audio could cost around $0.5 in the original implementation, and would cost only around $0.03 after applying the proposed fix.

Summary. The round-up manner of ML API charging policy creates yet another dimension into the already complicated trade-off space. Future work can extend program checkers and run-time controllers to consider economical effect as well.

VII. Solutions

We have implemented checkers and wrappers to automatically detect and fix some of the anti-patterns introduced in Section IV-VI. The auto-detection tools are implemented with Jedi [85], AST [86] and PyGitHub [87] library.

A. Output Misinterpretation Checker

We have built a static checker to automatically detect misuses of the `sentiment-detection` API's output, a type of accuracy bugs discussed in Section IV-B. Our checker first identifies every call site of the API, and then examines the data-flow graph to see whether both the `score` field and the `magnitude` field of the API result are used in later execution. Our analysis is inter-procedural and path sensitive. If the result is used as a parameter of a function call, we continue to check how/whether the result fields are used inside the callee function; if the result is returned by the current function, we continue to check how/whether the result fields are used in every caller function. The tracking ends either when we have confirmed that both fields, `score` and `magnitude`, have been used, or when we cannot see both of them being used after checking a threshold number of caller and callee functions. A bug is reported in the latter case.

Among the 975 GitHub Python applications that use this API, our checker finds 360 of them interpreting the API output incorrectly. We randomly sampled 30 detected bugs and only found one false positive: an application passes the API result to an html template to render a web page, which then uses both

Google Cloud Speech-to-Text

```
operation = client.long_running_recognize(config, audio)
result = operation.result()
```

AWS Transcribe

```
transcribe.start_transcription_job(...)
while True:
  status = transcribe.get_transcription_job(...)
  if status[...] in ['COMPLETED', 'FAILED']:
    break
  time.sleep(···)
```

Fig. 6: Using asynchronous API in synchronously (Blue lines contain key code structures used by our checker)

result fields. Unfortunately, HTML code analysis is currently not covered by our checker.

B. Asynchronous API call checker

As discussed in Section V-B, many applications in our benchmark suite call asynchronous APIs in a synchronous, blocking way, and hence suffer reduced performance for no benefit. To automatically identify this problem, our checker first identifies all the places where an asynchronous API is called and then the application immediately waits on the result, following the common API usage patterns shown in Figure 6. The checker then looks for other concurrent execution. If not, this pattern is tagged as a place for performance optimization.

To accurately identify code snippets that can execute concurrently with an asynchronous API call is difficult. Our checker examines if the function f calling the asynchronous API, or the callers of f, ever appears in the same Python file with any multi-thread and multi-process related Python APIs, in which case our checker conservatively thinks that f may be calling the asynchronous API concurrently with other execution in the program. Otherwise, this is reported as a performance problem.

Our checker is applied to 246 GitHub python applications using Google's Speech-to-Text asynchronous API, and reports 203 applications that issue at least one asynchronous call, while the caller blocks to wait for the result without other concurrent execution in the program. We manually checked 30 reported problems and found no false positives. Being conservative, our checker does have false negatives. For example, our manual checking finds that only 8 of the remaining 43 cases have called the asynchronous API in a concurrent way.

For 277 Python applications that use asynchronous AWS NLP and Speech APIs, our checker automatically reports 110 applications as having this type of performance problem. Our manual checking finds no false positives out of 30 randomly sampled problem reports. Note that, our checker may have more false negatives for AWS applications, as a number of applications use AWS Lambda auto-scheduler service [56] when making the asynchronous API call, which our checker conservatively assumes as having no performance problems.

C. Constant-parameter API call checker

We have implemented a static checker to automatically identify speech synthesis API calls that use constant inputs, a type of performance mis-use discussed in Section V-D. Our checker starts with every call site and tracks backward along the data dependency graph to see how the parameter of the API call is generated. Specifically, the checker keeps a working set that is initialized with the parameter itself p. It first identifies all the p assignments that can reach the API call site, and replaces p in the working set with all the non-constant variables at the right-hand side of those assignments. This back tracking continues until either (1) the working set becomes empty, in which case a constant-parameter API call problem is reported, or (2) our tracking has reached our inter-procedural checking threshold, configured as 5 levels of function calls, in which case we consider this API call as having a variable parameter.

We applied our checker to 686 (943) applications on GitHub that use Google's (AWS's) Python speech synthesis API. From them, our checker finds 202 (196) applications making the speech synthesis API calls with constant parameters. We then manually excluded those cases where the problematic calls are inside unit tests and, at the end, found 133 (146) applications having this performance problem inside their main program. By manually checking 60 reported applications, 30 each from AWS and Google, we found a total of 4 false positives. In 1 case, memoization is actually implemented; in the other 3 cases, a library call with constant parameters can actually return non-constant results, which confused our checker. Overall, as the number shows, this is really a widespread problem in machine learning applications.

D. API wrappers

We design API wrappers for all three domains of APIs. In vision tasks, our wrapper down-samples large images to the suggested size of 640×480 pixels. It tackles the anti-patterns of missing input validation (Section IV-C) and unnecessarily high-resolution inputs (Section V-E). In language tasks, the wrapper focuses on entity detection and syntax analysis, which allow input chunking with little impact to result accuracy. Our wrapper API takes in one or multiple text strings. It first concatenates all input strings together, which avoid the money wasting problem in Section VI. If the combined string is not too long, a synchronous API is called; if it is too long, it will be chunked and get processed through batching API, avoiding the anti-patterns of forgetting parallel APIs (Section V-C) and misuse of asynchronous APIs (Section V-B) . The wrapper for speech tasks is similar, but only takes one audio as input. The wrapper uses the synchronous API when the input size allows or streaming API otherwise. All these wrappers conduct an input validation and, in some cases, also transformation (Section IV-C).

The source code of all the checkers and wrappers is available online [50].

VIII. Threats to Validity

Internal threats to validity. The inputs used in our performance profiling and inference-accuracy measurement may not represent the exact workload used by real-world users. Our static checkers, as discussed in Section VII, can have false positives and false negatives.

External threats to validity. As discussed in Section III, we only studied ML APIs offered by Google and AWS in this work, but not those offered by other service providers. Our study only covers cloud APIs with pre-trained DNNs designed for general purpose use, and excludes user-defined DNNs based on their specific needs. We only study open-source projects on GitHub, with no access to those closed-source commercial projects. The 360 applications in our manual study benchmark suite may not represent all real-world applications. Our static analysis tool currently only covers python applications.

IX. Related Work

Prior work studies the different phases and different developer roles in large-scale development and deployment of ML-based applications [1], [88]–[90]. These applications design their own DNNs, instead of using existing ML APIs. Some work studies DNN deployment challenges caused by different frameworks and platforms and how to address them using techniques like DNN compression and quantization [91], [92].

Some research studies common mistakes in programs that design and train neural networks [16]–[19] or other types of machine learning models (e.g., SVM and decision tree) [93]. Some works focus on testing [20]–[45] and fixing [46]–[49] neural networks. All of these studies consider building machine learning models, instead of using them.

Another line of work focuses on the design and implementation of machine learning APIs, including machine learning frameworks like TensorFlow and PyTorch [9]–[15] and REST APIs for machine learning [94]–[96]. These works do not look at how ML APIs are used in larger software systems.

Much research has been done for designing and improving FaaS (Functions as a Service) platforms, in terms of performance [97]–[101], and security [102]–[104]. However, these works focus on the server side, instead of the client side. Some works [105]–[107] also examine the performance of enterprise FaaS platforms to help developers select service providers. Other works [108]–[110] aim to help developers move local computation to the cloud. These works improve application performance using general FaaS APIs, but do not address the unique challenges for ML APIs.

X. Conclusion

Cloud based machine learning APIs have become a popular approach for developers to leverage machine learning inference in software. This paper conducts a comprehensive study to understand the challenges in using these machine learning APIs. By investigating the latest versions of 360 open-source applications using Google and AWS ML Cloud APIs, we have found 8 types of common API misuses that cause functionality, performance, and service cost problems. We also develop static checkers to automatically detect some of these problems in a larger set of applications. The wide presence of these problems motivates future research to further tackle ML API misuses.

Data Availability

We have released our whole benchmark suite, automated checkers, and detailed study results online [50].

Acknowledgement

We thank the reviewers for their insightful feedback. The authors' research is supported by NSF (grants CCF-2028427, CNS-1956180, CCF-1837120, CNS-1764039, CNS-1563956, IIS-1546543, CNS-1514256), ARO (grant W911NF1920321), DOE (grant DESC0014195 0003), DARPA (grant FA8750-16-2-0004), the CERES Center for Unstoppable Computing, and the Marian and Stuart Rice Research Award.

References

[1] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *ICSE-SEIP*. IEEE, 2019, pp. 291–300.

[2] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009.

[3] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *ECCV*, 2014.

[4] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2012.

[5] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *CVPR*, 2016.

[6] R. Rothe, R. Timofte, and L. Van Gool, "Dex: Deep expectation of apparent age from a single image," in *ICCV*, 2015, pp. 10–15.

[7] M. Andriluka, L. Pishchulin, P. Gehler, and B. Schiele, "2d human pose estimation: New benchmark and state of the art analysis," in *CVPR*, 2014.

[8] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, T. Duerig *et al.*, "The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale," *arXiv preprint arXiv:1811.00982*, 2018.

[9] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *ICSE*, 2019.

[10] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler *et al.*, "Api design for machine learning software: experiences from the scikit-learn project," *arXiv preprint arXiv:1309.0238*, 2013.

[11] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning," in *ICDM*, 2013.

[12] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *arXiv preprint arXiv:1511.06435*, 2015.

[13] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *ASE*, 2019.

[14] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, L. Xiaohong, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *FSE*, 2020.

[15] S. Tizpaz-Niari, P. Cernỳ, and A. Trivedi, "Detecting and understanding real-world differential performance bugs in machine learning libraries," in *ISSTA*, 2020.

[16] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *ISSTA*, 2018, pp. 129–140.

[17] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *ESEC/FSE*, 2020.

[18] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *ICSE*, 2020.

[19] G. Jahangirova, N. Humbatova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE*, 2020.

[20] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *ASPLOS*, 2017.

[21] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *ICSE*, 2018.

[22] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *ISSTA*, 2019, pp. 146–157.

[23] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *ICML*, 2019.

[24] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: automated neural network model debugging via state differential analysis and input selection," in *ESEC/FSE*, 2018.

[25] S. Ma, Y. Aafer, Z. Xu, W.-C. Lee, J. Zhai, Y. Liu, and X. Zhang, "Lamp: data provenance for graph based machine learning algorithms through derivative computation," in *FSE*, 2017.

[26] N. D. Bui, Y. Yu, and L. Jiang, "Autofocus: interpreting attention-based neural networks by code perturbation," in *ASE*, 2019.

[27] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *ICSE*, 2018.

[28] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *ISSRE*, 2018.

[29] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *ASE*, 2018.

[30] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *ISSTA*, 2018.

[31] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: testing software for discrimination," in *FSE*, 2017.

[32] R. Angell, B. Johnson, Y. Brun, and A. Meliou, "Themis: Automatically testing software for discrimination," in *ESEC/FSE*, 2018.

[33] S. Amershi, M. Chickering, S. M. Drucker, B. Lee, P. Simard, and J. Suh, "Modeltracker: Redesigning performance analysis tools for machine learning," in *CHI*, 2015.

[34] S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, and X. Zhang, "Correlations between deep neural network model coverage criteria and model quality," in *ESEC/FSE*, 2020.

[35] F. Zhang, S. P. Chowdhury, and M. Christakis, "Deepsearch: A simple and effective blackbox attack for deep neural networks," in *ESEC/FSE*, 2020.

[36] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim, "Is neuron coverage a meaningful measure for testing deep neural networks?" in *ESEC/FSE*, 2020.

[37] V. Riccio and P. Tonella, "Model-based exploration of the frontier of behaviours for deep learning system testing," in *ESEC/FSE*, 2020.

[38] S. Gerasimou, H. F. Eniser, A. Sen, and A. Cakan, "Importance-driven deep learning system testing," in *ICSE*, 2020.

[39] B. Paulsen, J. Wang, and C. Wang, "Reludiff: Differential verification of deep neural networks," in *ICSE*, 2020.

[40] X. Zhang, X. Xie, L. Ma, X. Du, Q. Hu, Y. Liu, J. Zhao, and M. Sun, "Towards characterizing adversarial defects of deep learning software from the lens of uncertainty," in *ICSE*, 2020.

[41] D. Berend, X. Xie, L. Ma, L. Zhou, Y. Liu, C. Xu, and J. Zhao, "Cats are not fish: Deep learning testing calls for out-of-distribution awareness," in *FSE*, 2020.

[42] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks," in *ISSTA*, 2020.

[43] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *ISSTA*, 2020.

[44] S. Lee, S. Cha, D. Lee, and H. Oh, "Effective white-box testing of deep neural networks with adaptive neuron-selection strategy," in *ISSTA*, 2020.

[45] A. Sharma and H. Wehrheim, "Higher income, larger loan? monotonicity testing of machine learning models," in *ISSTA*, 2020.

[46] H. Zhang and W. Chan, "Apricot: a weight-adaptation approach to fixing deep learning models," in *ASE*, 2019.

[47] Z. Li, X. Ma, C. Xu, J. Xu, C. Cao, and J. Lü, "Operational calibration: Debugging confidence errors for dnns in the field," in *ESEC/FSE*, 2020.

[48] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *ICSE*, 2020.

[49] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *ICSE*, 2020.

[50] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "Project Webpage: Accurate Learning for EneRgy and Timeliness in Software System," https://alert.cs.uchicago.edu/#release.

[51] WhatsInYourFridge, "A smart fridge application," https://github.com/jitli98/whatsinyourfridge.

[52] Google, "Google cloud ai," Online document https://cloud.google.com/products/ai, 2020.

[53] Amazon, "Amazon artificial intelligence service," Online document https://aws.amazon.com/machine-learning/ai-services, 2020.

[54] IBM, "Ibm watson," Online document https://www.ibm.com/watson, 2020.

[55] Microsoft, "Microsoft azure cognitive services," Online document https://azure.microsoft.com/en-us/services/cognitive-services, 2020.

[56] Amazon, "Aws lambda," Online document https://aws.amazon.com/lambda/, 2020.

[57] M. Liwicki and H. Bunke, "Iam-ondb-an on-line english sentence database acquired from handwritten text on a whiteboard," in *ICDAR*, 2005.

[58] PDF2Text, "A pdf scanner application," https://github.com/CAU-OSS-2019/team-project-team06.

[59] Phoenix, "A fire-detection application," https://github.com/Flowmot1on/Phoenix.

[60] K. Lee, K. Lee, H. Lee, and J. Shin, "A simple unified framework for detecting out-of-distribution samples and adversarial attacks," in *NIPS*, 2018.

[61] Google, "Natural language api basics," Online document https://cloud.google.com/natural-language/docs/basics, 2020.

[62] JournalBot, "A journal application," https://github.com/beekarthik/JournalBot.

[63] AuthomaticDoor, "A smart door application," https://github.com/manuelleungchen/AuthomaticDoorSystem-Python-and-AWS-.

[64] Akamai and Gomez.com, "How loading time affects your bottom line," Online document https://blog.kissmetrics.com/loading-time/, 2011.

[65] A. LS, "What is simultaneous/conference interpretation," Online document, https://atlasls.com/what-is-simultaneousconference-interpretation/, 2010.

[66] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *ACM SIGCOMM*, 2018, pp. 253–266.

[67] Google, "Speech-to-text api basics," Online document https://cloud.google.com/speech-to-text/docs/basics, 2020.

[68] AnsweringMachine, "A bot application," https://github.com/devopsrebels/answeringmachine.

[69] S. Kakar, "How to create a sales cadence that doesn't annoy prospects," Online document https://www.copper.com/blog/sales-cadence/, 2019.

[70] JiangJungDian, "A meeting management application," https://github.com/mre500/jiang-jung-dian.

[71] G. Cloud, "Transcribing audio from streaming input," https://cloud.google.com/speech-to-text/docs/streaming-recognize.

[72] PottyPot, "A real-time audio analysis application," https://github.com/BlakeAvery/PottyPot.

[73] Class-Scribe-LE, "A lecture note application," https://github.com/rahatmaini/Class-Scribe-LE.

[74] Okuninushi, "A web application for japanese sake," https://github.com/BlackWinged/Okuninushi.

[75] EmailClassifier, "An email classification application," https://github.com/Kalsoomalik/EmailClassifier.

[76] R. Weischedel and A. Brunstein, "Bbn pronoun coreference and entity type corpus," *Linguistic Data Consortium, Philadelphia*, 2005.

[77] R. Weischedel, S. Pradhan, L. Ramshaw, M. Palmer, N. Xue, M. Marcus, A. Taylor, C. Greenberg, E. Hovy, R. Belvin *et al.*, "Ontonotes release 4.0," *LDC2011T03, Philadelphia, Penn.: Linguistic Data Consortium*, 2011.

[78] Samaritan, "A medical document analysis application," https://github.com/edmondchensj/samaritan-backend.

[79] J. Yamagishi, C. Veaux, K. MacDonald *et al.*, "Cstr vctk corpus: English multi-speaker corpus for cstr voice cloning toolkit (version 0.92)," 2019.

[80] Google, "Google audioset: A large-scale dataset of manually annotated audio events," https://research.google.com/audioset/.

[81] S. O. Runeterra, "An card came extension for visually impaired gamers," https://github.com/AlejandroCabeza/sounds_of_runeterra.

[82] NsTool, "A monitor application," https://github.com/clarkwkw/ns_online_toolkit.

[83] Tags, "A video scene clustering application," https://github.com/OsamaAl-Wardi/Tags.

[84] AudioSentenceSplit, "A speech recognition application," https://github.com/ynotnplol/Audio-SentenceSplit.

[85] D. Halter, "Jedi: an awesome auto-completion, static analysis and refactoring library for python," Online document https://jedi.readthedocs.io.

[86] Python, "ast — abstract syntax trees," https://docs.python.org/3/library/ast.html.

[87] PyGithub, "Pygithub: Typed interactions with the github api v3," https://pygithub.readthedocs.io/en/latest/introduction.html.

[88] C. Hill, R. Bellamy, T. Erickson, and M. Burnett, "Trials and tribulations of developers of intelligent systems: A field study," in *VL/HCC*, 2016.

[89] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "The emerging role of data scientists on software development teams," in *ICSE*, 2016.

[90] ——, "Data scientists in software teams: State of the art and challenges," *TSE*, 2017.

[91] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *ASE*, 2019.

[92] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A comprehensive study on challenges in deploying deep learning based software," in *ESEC/FSE*, 2020.

[93] Y. Tao, S. Tang, Y. Liu, Z. Xu, and S. Qin, "How do api selections affect the runtime performance of data analytics tasks?" in *ASE*, 2019.

[94] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are rest apis for cloud computing well-designed? an exploratory study," in *ICSOC*. Springer, 2016, pp. 157–170.

[95] E. Gossett, C. Toher, C. Oses, O. Isayev, F. Legrain, F. Rose, E. Zurek, J. Carrete, N. Mingo, A. Tropsha *et al.*, "Aflow-ml: A restful api for machine-learning predictions of materials properties," *Computational Materials Science*, 2018.

[96] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential regression testing for rest apis," in *ISSTA*, 2020.

[97] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *ICDCSW*, 2017.

[98] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *IC2E*, 2018.

[99] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *UCC Companion*, 2018.

[100] J. Kijak, P. Martyna, M. Pawlik, B. Balis, and M. Malawski, "Challenges for scheduling scientific workflows on cloud functions," in *CLOUD*, 2018.

[101] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *CLOUD*, 2019.

[102] M. S. Ferdous, A. Margheri, F. Paci, M. Yang, and V. Sassone, "Decentralised runtime monitoring for access control systems in cloud federations," in *ICDCS*, 2017.

[103] J. Kim, J. Park, and K. Lee, "Network resource isolation in serverless cloud function service," in *FAS* W*, 2019.

[104] A. T. Gjerdrum, H. D. Johansen, L. Brenna, and D. Johansen, "Diggi: A secure framework for hosting native cloud functions with minimal trust," in *TPS-ISA*, 2019.

[105] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *CloudCom*, 2017.

[106] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *CLOUD*, 2018.

[107] I. Pelle, J. Czentye, J. Dóka, and B. Sonkoly, "Towards latency sensitive cloud native applications: A performance study on aws," in *CLOUD*, 2019.

[108] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, and V. Yussupov, "Modeling and automated deployment of serverless applications using tosca," in *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2018, pp. 73–80.

[109] W. Lloyd, M. Vu, B. Zhang, O. David, and G. Leavesley, "Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads," in *UCC Companion*, 2018.

[110] J. Scheuner and P. Leitner, "Transpiling applications into optimized serverless orchestrations," in *FAS* W*, 2019.