THE UNIVERSITY OF CHICAGO

# CHARACTERIZING, DETECTING, AND EXPOSING MIS-USES OF MACHINE LEARNING CLOUD APIS IN OPEN-SOURCE SOFTWARE

A THESIS SUBMITTED IN PARTIAL SATISFACTION
OF THE REQUIREMENTS FOR THE HONORS DEGREE OF

BACHELOR OF COMPUTER SCIENCE

IN THE

DEPARTMENT OF COMPUTER SCIENCE

BY

SHICHENG LIU

CHICAGO, ILLINOIS

JUNE 2022

# TABLE OF CONTENTS

# ACKNOWLEDGMENTS

First and foremost, I would like to express my special thanks of gratitude to my advisor Shan, whom I had the great honor to work with starting from winter of my second year. Shan is an excellent researcher, a wonderful team leader, and above all, a great mentor.

I would also like to thank other members of the research group whom I had the opportunity to work with. Chengcheng would be on the top of that list.

It also goes without saying that I am forever in debt of my parents, for their support and everlasting love.

Finally, I would also like to thank my friends, at UChicago and beyond. My 4 years in college have been absolutely wonderful, thanks to you all!

# ABSTRACT

Machine Learning (ML) cloud APIs - an increasingly popular product - enable developers to easily incorporate learning solutions into software systems. Prior work has studied developing ML models and cloud services, but no comprehensive study on how software developers are using these products has been conducted. In the series of work [1] [2] I co-author, we characterize ML cloud API mis-uses in open-source software, generalized from the detailed study of representative applications. We further develop tools, including static checkers and novel testing algorithms, to detect and expose these mis-uses.

The main contributions of this series of works include: (1) A comprehensive empirical study on 360 representative open-source applications that use Google or AWS cloud-based ML APIs, and - as a result - a characterization of mis-uses that negatively impact the functionality, performance, or cost-efficiency of around 70 % of these software; (2) Developments of static analyzers which successfully detect relevant performance-related problems among around 500 Github applications; (3) A novel, comprehensive testing algorithm which successfully exposes critical functionality-related problems, available as a VS Code IDE plugin for relevant Google Cloud APIs.

# CHAPTER 1
# INTRODUCTION

In this section, this thesis will provide a general discussion on ML cloud APIs, their current usages in open-sourced software, and a pointer to the current state of research on ML cloud APIs.

## 1.1 Background & Overview

Machine Learning (ML) serves as an efficient solution for various tasks that are difficult to solve with traditional computing techniques. Examples of its success can be seen across domains in natural language processing (e.g. language translation), computer vision (e.g. object classification), and speech processing (e.g. voice assistant), to name a few. However, training these learning models requires expert knowledge, and the research of the state-of-the-art models is an active area in itself. In bridging this gap between top-tier ML models and their ordinary usages in software systems, a number of products in ML cloud services has emerged recently. These include, for instance, Google Cloud AI [3], Amazon Web Service (AWS) AI [4], IBM Cloud Watson [5], and Microsoft Azure Cognitive Services [6]. These products package the usage of pre-trained, deep neural networks (DNNs), i.e. ML solutions, into cloud APIs calls, thus enabling software developers to utilize these powerful solutions freed from the hindrance of the expert knowledge required for training models and resource allocation. These products span a variety of common application domains (as discussed in Section 1.2) and are gaining popularity (as discussed in Section 1.3). To help the readers better understanding the usages of these APIs, two example workflows are provided in Section 1.4.

While a variety of prior researches has been dedicated to the study of ML libraries, training of ML models, and design of cloud services (as discussed in Section 1.5), this thesis focuses on the unique challenges posed by the ML nature of these APIs. This leads to the identification of anti-patterns *specific to ML cloud APIs* (as discussed in Chapter 2), among which the two categories of most interest are: (1) performance-related anti-patterns (Section 2.3) and (2) functionality-related anti-patterns (Section 2.4). We propose solutions in the form of static analyzers (Chapter 3) and testing algorithms (Chapter 4) to detect and expose these problems, respectively.

## 1.2 Major Functionalities of ML cloud APIs

Common application domains of the aforementioned four major ML cloud API service providers mainly include the following:

1. **Vision**: This includes image-oriented and video-oriented ML processing, e.g., detecting objects, faces, landmarks, logos, text, or sensitive content from an image or a video;
2. **Language**: This is commonly referred to as Natural Language Processing (NLP) tasks and include, e.g., analyzing and detecting entities, sentiments, translating languages, and syntax from natural language text inputs;
3. **Speech**: This includes text-to-speech synthesis (synthesizing an audio from text inputs) and speech-to-text transcription (recognizing text from an audio input).

Table 1.1 summarizes the main available services. One can notice that certain tasks are only supported with synchronous APIs, and certain tasks are only supported with asynchronous APIs, while the rest are supported with both. For the services that both forms of APIs are available, it poses complicated input-accuracy-performance tradeoffs, which proves difficult for developers to navigate. This tradeoff and the consequent anti-pattern will be discussed in detail in Section 2.3.1.

|  |  | Google Cloud AI | AWS AI |
|---|---|---|---|
| Vision | Image | Vision AI | Rekognition |
|  | Video | Video AI |  |
| Language | NLP | Cloud Natural Language $_{\mathbf{S}}$ | Comprehend |
|  | Translation | Cloud Translation $_{\mathbf{S}}$ | Translate $_{\mathbf{S}}$ |
| Speech | Recognition | Speech-to-Text | Transcribe $_{\mathbf{A}}$ |
|  | Synthesis | Text-to-Speech $_{\mathbf{S}}$ | Polly |

|  |  | IBM Cloud Watson | Microsoft Azure |
|---|---|---|---|
| Vision | Image | Visual Recognition $_{\mathbf{S}}$ | Computer Vision, Face |
|  | Video | - | Video Indexer $_{\mathbf{A}}$ |
| Language | NLP | Natural Language Understanding $_{\mathbf{S}}$ | Text Analytics |
|  | Translation | Language Translator | Translator |
| Speech | Recognition | Speech to Text | Speech to Text |
|  | Synthesis | Text to Speech $_{\mathbf{S}}$ | Text to Speech |

Table 1.1: ML tasks supported by four popular ML cloud services. Subscript $_{\mathbf{S}}$: only a synchronous API is offered for this task; subscript $_{\mathbf{A}}$: only an asynchronous API is offered; no subscript: both synchronous and asynchronous APIs are offered.

## 1.3 Increasing Popularity of ML cloud APIs

In this section, we provide statistics on the number of GitHub applications using such ML cloud API services. Our work on characterizing relevant mis-uses (Section 2) focuses mostly on Google Cloud AI and AWS AI services, and the number of open-source GitHub applications using them is reported in Table 1.2. These numbers were latest as of 08-27-2020, among which a total of (without duplicates across services) 14049 were created after 08-01-2019.

These statistics were produced from a Ruby script that queries GitHub search console using respective API language-dependent function call name. Because of possible overlapping function call names with other non-related functions, we experiment to find out the most accurate query keywords in maximizing true positives and minimizing false positives. For instance, table 1.3 details the keywords used in searching for language translation, speech-to-text, and text-to-speech Google cloud APIs. The keywords used for the remaining Image, Video, and NLP tasks (as displayed in Table 1.2) are consistent across all languages and are: `ImageAnnotatorClient`, `VideoIntelligenceServiceClient`, and

|  |  | All Apps |  |
|---|---|---|---|
|  |  | Google | AWS |
| Vision | Image | 7916 | 8818 |
|  | Video | 674 |  |
| Language | NLP | 4632 | 4291 |
|  | Translation | 1192 | 7681 |
| Speech | Recognition | 9439 | 5155 |
|  | Synthesis | 2190 | 6375 |
| Total (w/o duplicates) |  | 35376 |  |

Table 1.2: Number of applications using different types of Google and AWS ML cloud APIs on GitHub.

|  | Translation | Recognition | Synthesis |
|---|---|---|---|
| Python | google.cloud\s TranslationServiceClient | google.cloud\s SpeechClient | TextToSpeechClient |
| PHP | Google\Cloud\s TranslationServiceClient | Google\Cloud \SpeechClient |  |
| Java | google.cloud\s TranslationServiceClient | google.cloud. SpeechClient |  |
| Node.js | google-cloud\s TranslationServiceClient | google-cloud/speech |  |
| C# | TranslationServiceClient. Create | Google.Cloud.Speech |  |
| Go | translatepb | cloud.google speech | texttospeechpb |
| Ruby | google/cloud/translate | google/cloud/speech | Google::Cloud ::TextToSpeech |

Table 1.3: Keywords used in retrieving the number of GitHub applications using different types of Google cloud APIs on GitHub. For space-saving purposes, the '\s' character is used to denote a space character, and there is no newline character used in the actual searches.

`LanguageServiceClient`. Thus, due to these keyword choices, this method may not produce 100 % accuracte search results, but should serve as a relatively accurate estimate for the relevant statistics.

These numbers illustrate the relatively young age of open-source applications using ML cloud APIs and their increasing popularity. Thus, a comprehensive empirical study needs to be conducted to understand how developers are using them in open-source applications. A survey of related works will be provided in Section 1.5.

## 1.4 Exemplary Workflows

To help the readers understand how exactly these cloud APIs are used, this section will provide the workflows of two representative GitHub open-sourced applications.

Figure 1.1(a) illustrates the general workflow of **Whats-In-Your-Fridge** [7], an application for recipe suggestion. This application uploads a photo taken inside a fridge to the cloud, calls a cloud vision API to find out what objects are inside the fridge, and generates recipes based on the results from vision API.

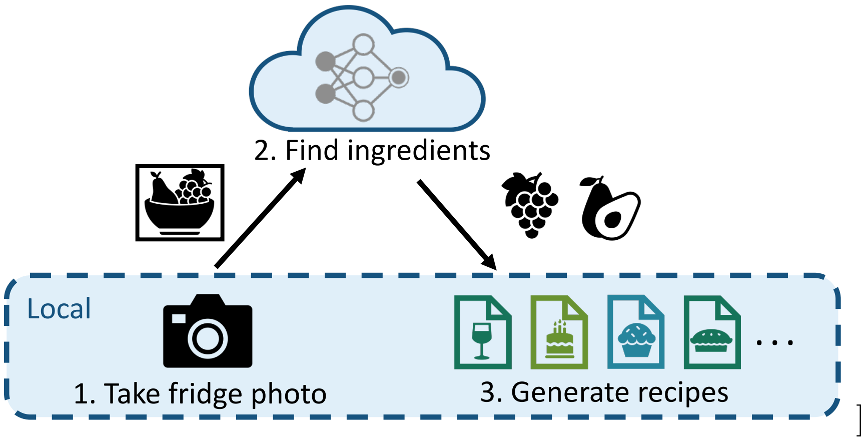
Figure 1.1(b) illustrates the general workflow of **Pheonix** [8], an application for fire detection. This application feeds a photo to Google Cloud's image classification API and checks the returned labels to identify whether a fire is ongoing. If so, it raises an alarm.

More example applications will be provided throughout this thesis. In general, we are interested in applications that use the ML cloud API outputs in meaningful and non-trivial ways, as in these two examples. More information on which applications are selected for our study is discussed in Section 2.2.

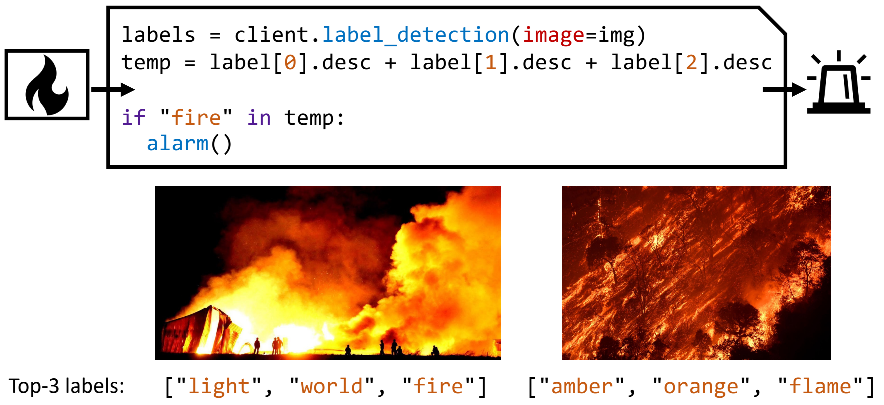
## 1.5 Overview of Prior Related Works

**Machine Learning and Deep Neural Networks**: A significant amount of research has evolved out of the need to perfect ML models. More recently, deep neural networks have emerged as a promising solution for many ML tasks. In this research, we do not intend to design or train models. Instead, we focus on improving applications of one particular usage of these deep learning models (ML cloud APIs).

**Testing Machine Learning Models**: There has been an emerging focus in the Software Engineering and related communities for testing and fixing bugs in neural networks. For example, there has been various testing and verification proposals [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22] designed specifically for neural network models. This line of work



(a) Example in **Whats-In-Your-Fridge** [7]



```
labels = client.label_detection(image=img)
temp = label[0].desc + label[1].desc + label[2].desc

if "fire" in temp:
  alarm()
```

(b) Example in **Phoenix** [8]

Figure 1.1: Examples of using ML cloud APIs

is needed because traditional testing techniques do not apply directly to neural networks, which pose separate challenges regarding its testing and verification. This line of work differs from our research because their primary targets are the machine learning models themselves, as opposed to software using machine learning APIs. Again, our work does not aim to test machine learning models. Instead, our focus is on the software employing machine learning models in the form of cloud APIs.

**Developing Machine Learning Libraries and APIs**: Similar to the need for testing machine learning models, it is also important to design better implementation for machine learning libraries and APIs (e.g. PyTorch and TensorFlow). Relevant works include ML framework design and implementation [23, 24, 25] and testing proposals for these frameworks [26, 27]. This line of work, similar to the above, differs from ours since they do not focus on higher-level programs utilizing ML libraries (and ML cloud APIs in particular).

# CHAPTER 2
# CHARACTERIZATION OF MIS-USES OF ML CLOUD APIS

In this chapter, this thesis will detail the anti-patterns discovered in our empirical study, spanning performance-related, functionality-related, and cost-related domains, along with a discussion on our methodology and benchmark applications.

## 2.1 Methodology

Since these applications using ML cloud API are relatively young (see Section 1.3 and Section 2.2), we cannot simply rely on known API misuses reported in their issue-tracking systems, which are very rare. Instead, it is vital to discover anti-patterns unknown to the developers through our own efforts. Further, to our knowledge, there has not been a similar systematic, empirical study focused on this set of APIs, meaning that we also cannot rely on any existing list of anti-patterns.

Thus, in our research, our team - which includes ML researchers - carefully studies API manuals, intensively profiles the API functionality and performance, and then manually examines every use of a ML cloud API in our benchmark applications (Section 2.2) for potential misuses. For every suspected misuses, we design test cases and run the corresponding application or component to understand whether it actually leads to reduced functionality, degraded functionality, or increased cost compared to an alternative usage of the same API devised by us.

Once one misuse is confirmed, we attempt to generalize it and check if there are similar misuses in other relevant applications. This process of identification-confirmation-generalization is repeated multiple rounds. In the process, we also report representative misuses to corresponding application developers and have received multiple acknowledgements, as discussed in detail in the next few sections.

## 2.2 Selection of Benchmark Applications

We collect a benchmark suite of 360 non-trivial applications that use Google or AWS ML cloud APIs - 120 for each major domain (vision, language, and speech, as shown in Table 1.2). They include applications mainly written in Python (80%), JavaScript (13%), Java (3%), and others (4%). Around 80% use API from Google Cloud AI and 20% use API from AWS, with 1% using both. The reason we include fewer applications using AWS is due to the popular usage of AWS Lambda [28] in these applications. AWS Lambda is a serverless computing platform which makes precise analysis of application workflow difficult. The sizes of these applications range from 46 to 3 million lines of code, with 2228 lines of code being the median size and around 40% having more than more than 10 thousand lines of code. 98% of these applications are created after 2018. We believe the relatively young age of these applications reflect that incorporating machine learning solutions (in particular, deep learning solutions, in the form of cloud APIs) into day-to-day software has only been recently recognized and yet is being adopted with unprecedented pace and breadth.

It is also of interest to remark on how these benchmark applications are selected. Since there are many toy applications on GitHub using these ML cloud APIs (e.g., printing out returned results from APIs, storing API results in some local file), we manually check about 1200 randomly selected applications that use Google our AWS ML cloud APIs, eventually obtaining these 360 applications. We manually confirm that they each target a concrete real-world problem domain, integrate ML APIs in their workflow, and conduct some pre- and post-processing of ML API outputs.

## 2.3 Performance-related Anti-patterns

This section will discuss the identified performance-related anti-patterns (corresponding to rows highlighted in blue in Table 2.2) in detail. All profiling experiments are conducted

| Related APIs and Inputs | Service Provider | Impact | # (%) of Problematic Apps. | |
|---|---|---|---|---|
| | | | Manual | Auto |
| **Should Have Called a Different API** | | | | |
| 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%) | - |
| 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%) |
| 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** | | | | |
| Speech Synthesis APIs with constant inputs | AG | Slower, More Cost | 15 ( 25%) | 279 (17%) |
| Vision-Image APIs with high call frequency | AG | Slower, More Cost | 3 ( 3%) | - |
| **Should Have Converted the Input Format** | | | | |
| all APIs without input validation, transformation | AG | Exceptions | 206 ( 57%) | - |
| Vision-Image APIs with high resolution inputs | AG | Slower | 106 ( 88%) | - |
| 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** | | | | |
| sentiment-detection | G | Low Accuracy | 24 ( 39%) | 360 (37%) |
| Total number of benchmark applications with at least one API misuse | AG | | 249 (69%) | |

Table 2.1: 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.

on the same machine, which contains a 16-core Intel Xeon E5-2667 v4 CPU (3.20GHz), 25MB L3 Cache, 64GB RAM, and 6$\times$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.

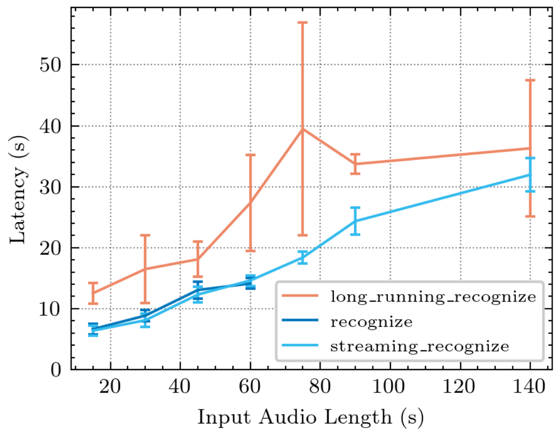
The profiling experiments discussed in this section report the end-to-end latency for each module (e.g., input processing, ML cloud API call) and also for the whole process. We use real-world vision, audio, or text input data that fit the application's working scenario. Unless otherwise specified, each data point is the average latency across 5 profiling experiments.

### *2.3.1 Mis-uses of Asynchronous APIs*

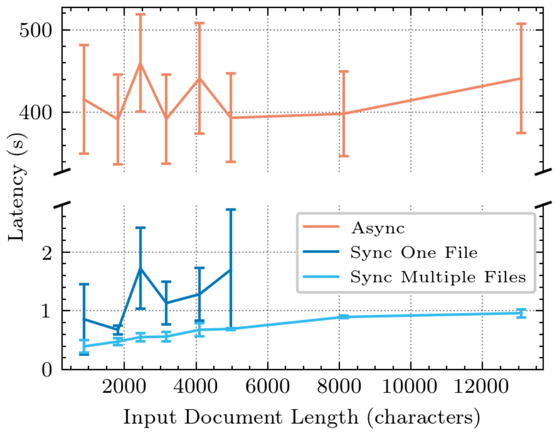
The ML cloud API providers often provide multiple APIs - a synchronous version, an asynchronous version, and possibly a streaming version (see Table 1.1) - for the same functionality. We discover that a common problem arises out of the usage of asynchronous APIs. In many traditional concurrent programs, asynchronous functions are used to gain performance through improved concurrency at the cost of extra software development efforts. However, in the case of these ML cloud APIs, we see an opposite tradeoff: asynchronous APIs are called without improved performance in exchange for less effort in input transformation.

**More lenient input requirements**: The benefit of asynchronous APIs is clearly documented: they allow much longer audio or text inputs than their synchronous counterparts. For example, in Google speech recognition service [29], the synchronous API takes audio up to 1-minute long, while the asynchronous API can take up to 480 minutes. A streaming API is also provided with no limit on input sizes. Another example in Natural Language processing is AWS Comprehend [30], which offers three APIs, with corresponding input limits in parenthesis: synchronous API (5000 bytes, 1 document), synchronous batching API (5000 bytes per document, 25 documents) and asynchronous API (no limit).

**(Much) worse performance**: However, the performance downside of these asyn-



(a) Google Speech-to-Text



(b) AWS Comprehend

Figure 2.1: Latency profiling for three different Python APIs of Google Speech-to-Text (synchronous, asynchronous, and streaming) and AWS Comprehend entity detection (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.

chronous APIs is not quantitatively documented. 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 2.1(a)[1]. The difference is even more extreme for AWS Comprehend Service in the Natural Language Processing domain: a synchronous API is 400X faster than an asynchronous version, as shown in Figure 2.1(b)[2] [3].

We discover that most applications use asynchronous API in a synchronous fashion. Namely, the caller to the asynchronous API would block itself until the API returns, with no other concurrent execution on going. This implementation implies that there is no way to compensate for the poor performance of asynchronous APIs. 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, which will be

1. Profiled on three different inputs: a news broadcast, and online lecture, and a WSJ audio

2. The function call names for synchronous one file, synchronous multi-file, and asynchronous API are respectively `detect_entities`, `batch_detect_entities`, and `start_entities_detection_job`.

3. Profiled on three different inputs: a philosophy text, a novel with conversations, and a CNN news article

discussed in detail in Chapter 3, confirms this trend: 203 out of 246 GitHub applications call this asynchronous API in a synchronous way.

**Replace with synchronous API call**: Since synchronous API run much faster than asynchronous API (Figure 2.1), one refactoring technique is to replace asynchronous API calls with synchronous API calls for many of these applications. For applications that expect most of their inputs to satisfy the synchronous API input size requirement, it is desirable to insert a condition check on the size of the API input: if the size fits into the synchronous API, then issue a synchronous API call; otherwise, proceed with the asynchronous API. More advanced techniques can be devised to further boost performance. For instance, chopping inputs into multiple parts and feeding them to the synchronous API could be a more advanced solution. Here, we illustrate the effectiveness of the simple refactoring technique with two benchmark applications, whose inputs mostly fall into the synchronous API size requirement.

1. **Answering-Machine** [31] applies the Google 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 [32], 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.

2. **Jiang-Jung-Dian** [33] is an application that automatically generates meeting reports. As demonstrated in Figure 2.2, its workflow goes as follows: (1) record individual voices and names of speakers who will speak in a meeting; (2) extract certain voice traits to associate voice with speaker name and save to disk; (3) record audio from a meeting; (4) upload the audio to cloud server; (5) use AWS Transcribe service with asynchronous API to get text transcription of meeting and wait until AWS Transcribe finishes; (6)

| **Stages of Workflow** | **Latency (s)** |
|---|---|
| Record a meeting | 59.8 |
| Upload file | 0.7 (± 0.02) |
| Async Speech2Text | 78.5 (± 6.46) |
| Download result & clean up | 0.4 (± 0.13) |
| Async Comprehend | 410.2 (± 55.88) |
| Sum | 549.6 (± 55.21) |

Table 2.2: Latency profile for key steps of **Jiang-Jung-Dian**

download result to local computer; (7) AWS Transcribe will return a list of audio staring and ending times with which speaker is speaking at what time, chunk the meeting audio file by speaker based on this result; (8) use the association from step (2) to determine the name of the speaker at the respective times chunked at step (7), generating person-by-person transcripts; (9) use AWS Comprehend service (asynchronous API) to detect entities from the entire meeting voice message, and (10) generate a UI report. It needs around 10 minutes to process a one-minute meeting. As shown in our profiling results in Table 2.2, most of the time is spent on two asynchronous ML cloud API calls offered by AWS. In particular, the asynchronous text Comprehend API call alone takes close to 7 minutes. 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.

**Replace with streaming call**: In addition to a synchronous and asynchronous version, some APIs also support a streaming version designed for real-world audio content that takes a streaming form. The Google Cloud speech recognition service [29], for instance, provides such a streaming API (AWS offers streaming APIs but not in the Python SDK). This Google recognition API can be either applied to local inputs, as in the setting for Figure 2.1 [4], or

4. This could seem counter-intuitive, but Google Cloud provides an explicit tutorial for using streaming API on local files. In such scenario, the audio stream generator is simply the local file itself. It also instructs: "While you can stream a local audio file to the Speech-to-Text API, it is recommended that you perform synchronous or asynchronous audio recognition for batch mode results."[34]

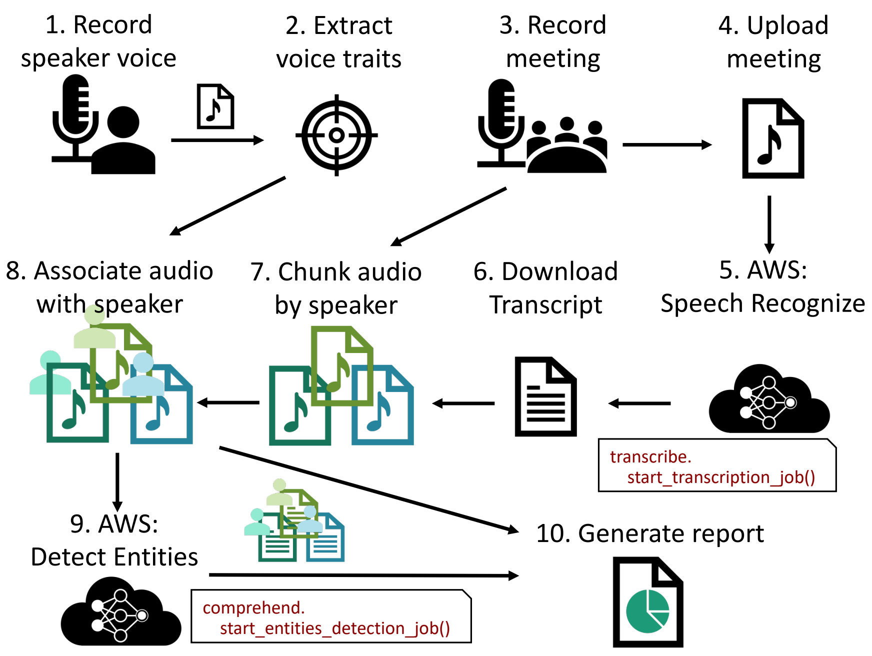


Figure 2.2: Workflow of application **Jiang-Jung-Dian**[33].

to a streaming input.

These APIs 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 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. Here are some examples of this refactoring:

3. **Potty-Pot** [35] 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.

4. **Class-Scribe-LE** [36] 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.

### *2.3.2 Making Skippable API Calls*

For certain API calls, it is unnecessary to continue issuing repeated API calls on constant inputs that can be skipped. This can be accomplished at the cost of slightly higher engineering effort while incurring indiscernible functionality difference. In the Speech Synthesis API, we discovered that among the 60 benchmark applications in this category, 15 (25%) of them call this API with constant string input and thus could have replaced the API call with a pre-recorded audio from a one-time API call. Our automated checkers (more in Chapter 3) further confirm this trend. We illustrate this refactoring technique with one benchmark application:

5. **Sounds-Of-Runeterra** [37] (Figure 2.3) is 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.

```
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)
    ...
```

Figure 2.3: Skippable call@ **Sounds-Of-Runeterra**[37]

### *2.3.3 Forgetting Parallel API Calls*

Certain services provide a parallel API for task parallelism (i.e. an API completing multiple tasks with the same function call) and data parallelism (i.e. an API processing multiple files with the same function call). For applications whose working scenarios fit these parallel APIs, an easy switch to these APIs would significant boost performance.

**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. When the application needs to call multiple ML APIs with no dependency among them, the calls should definitely be made in parallel—since these APIs are executed in the cloud, the parallelism will offer speedups without putting any extra resource contention to the local machine. 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:

6. Okuninushi[38], 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. This more advanced optimization provides appealing performance speedup, as illustrated in the examples below with different techniques.

7. **EmailClassifier** [39] 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 [40, 41]. 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.

8. **Samaritan** [42] 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 [43] mostly consists of 2-6 second audio clips, and Google Audioset [44] 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.

## 2.4 Functionality-related Anti-patterns

In addition to the performance-downgrading anti-patterns identified above, we also generalize 3 main types of API misuses that deteriorate the functional correctness of applications. These are listed as white-background rows in Table 2.2.

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.

This thesis will discuss one type of these mis-uses - *Calling the wrong API* (first three rows in Table 2.2). The other two types are outside the scope of this thesis and are discussed in [1] in detail.

Functionality-related mis-uses are significant for ML cloud APIs because of their machine learning nature: Unlike traditional APIs that are programmed to each conduct a clearly-coded, well-defined 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.

**`Text-detection` and `document-text-detection`**: They 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 [45] 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:

9. **PDF-to-text** [46] 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`**: They 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:

10. **Whats-In-Your-Fridge** [7] 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.

11. **Phoenix** [8] 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.

**`sentiment-detection` and `entity-sentiment-detection`**: They are both language APIs that 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.

## 2.5 Cost-related Anti-patterns

Additional cost-related anti-patterns are highlighted in yellow in Table 2.2. This thesis shall refer readers to [1] for further information on those anti-patterns.

# CHAPTER 3
# Q: HOW TO DETECT PERFORMANCE-RELATED PROBLEMS? A: STATIC ANALYZERS

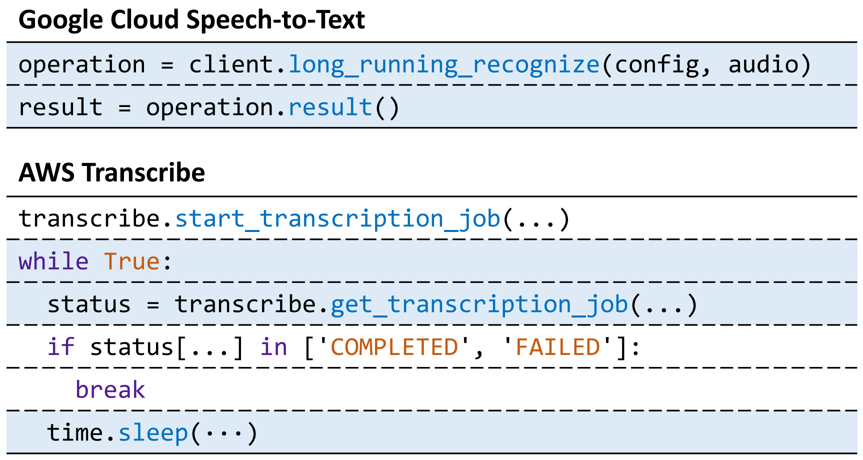
In this chapter, this thesis will discuss the solutions to performance-related problems identified in 2.3 in the form of two static analyzers. We deploy these static analyzers to relevant applications hosted on GitHub and automatically identify around 500 applications containing these API mis-uses.

## 3.1 Detection of Mis-uses of Asynchronous APIs

As discussed in Section 2.3.1, a lot of applications in our benchmark suite are misusing asynchronous APIs, by using them in a synchronous fashion. Hence, these applications suffer from reduced performance at no extra benefits. Example mis-uses are provided in Figure 3.1 for Google Cloud Speech-to-Text and AWS Transcribe services. We identify that there is an opportunity for an automated static analyzer to detect the same issue in similar applications.

The algorithm for this static analyzer is presented in Algorithm 1 and discussed here. To automatically identify this problem, our checker first identifies all the places where an asynchronous API is called (line 1) and then the application immediately waits on the result, following the common API usage pattern shown in Figure 3.1 (line 2). The checker then looks for other concurrent execution. If not, this pattern is tagged as a place for performance optimization (line 7-9).

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



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

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

Figure 3.1: Using asynchronous API in synchronously (Blue lines contain key code structures used by our checker)

related Python APIs[1], in which case our checker conservatively thinks that $f$ may be calling the asynchronous API concurrently with other execution in the program and returns a (mis-uses) `NOT_IDENTIFIED` flag as output (line 11). Otherwise, this is reported as a performance problem (line 7-9).

**Algorithm 1** Detect mis-uses of asynchronous APIs

```
Input: repo (a repository using asynchronous API)
Output: misue (whether the asynchronous API is mis-used)
1: for code_line contains async API do
2:   if code_line follows pattern as in Figure 3.1 then
3:     $G \leftarrow$ function call graph of API in code_line
4:     if $G$ contains AWS Lambda then
5:       return $misuse$ = NOT_IDENTIFIED
6:     end if
7:     if $G$ does not contain multi-thread feature then
8:       return $misuse$ = NO_PARALLELISM
9:     end if
10:  else
11:    return $misuse$ = NOT_IDENTIFIED
12:  end if
13: end for
14: return $misuse$ = USE_PARALLELISM
```

1. These libraries include `threading`, `multiprocessing`, `concurrent`, `asyncio`, `_thread` in this experiment.

Some applications uses AWS Lambda service, as mentioned in Section 2.3.1, which makes the precise identification of function call graph difficult. In such cases, our checker conservatively returns a (mis-uses) `NOT_IDENTIFIED` flag as output (line 4-6).

The analyzer will return a `USE_PARALLELISM` flag if, after checking all function calls to asynchronous APIs, all conditions mentioned above are met. This corresponds to line 14 in Algorithm 1.

| **Service** | **Google Cloud** | **AWS** |
|---|---|---|
| | Speech2Text | Speech2Text & Text2Speech & NLP |
| `NO_PARALLELISM` | 203 | 110 |
| `NOT_IDENTIFIED` | 31 | 153 |
| `USE_PARALLELISM` | 12 | 14 |
| Total | 246 | 277 |

Table 3.1: Output reported by asynchronous API mis-use static analyzer (Algorithm 1). The static analyzer errors on the side of caution and reports only the `NO_PARALLELISM` (in gray) as problematic. We expect certain false negatives among the applications flagged as `NOT_IDENTIFIED` and `USE_PARALLELISM` (in cyan).

The results of our static analyzer is reported in Table 3.1. 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 (reported as 31 + 12, the sum of the numbers in cyan, in Table 3.1) have called the asynchronous API in a concurrent way.

For 277 Python applications that use asynchronous AWS NLP, Speech Recognition, and Speech Synthesis APIs, our checker automatically reports 110 applications as having this type of performance problem (Table 3.1). Our manual checking finds no false positives out of 30 randomly sampled programs reported as `NO_PARALLELISM`.

## 3.2 Detection of Repeated, Skippable API Calls

As discussed in 2.3.2, many applications would repeatedly call speech synthesis cloud API with a fixed input. Caching the result and using the same audio from a single cloud API call will increase performance while incurring no distinguishable functionality difference.

We have implemented a static checker (Algorithm 2) to automatically identify this performance anti-pattern. Our checker starts with every call site (line 1) 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 $p$ (line 2-3). 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 (line 6, 10, 11). This back tracking continues until either (1) the working set becomes empty, in which case a constant-parameter API call problem is reported (line 7-9), 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 (line 4, 5, 14).

**Algorithm 2** Detect constant inputs to speech synthesis API

```
Input: repo (a repository using speech synthesis API)
Output: misuse (whether the speech synthesis API is mis-used)
 1: for code_line containing speech synthesis API do
 2:     p ← empty set
 3:     initialize p with inputs to speech synthesis API at code_line
 4:     procedure_counter = 0
 5:     while procedure_counter ≤ 5 do
 6:         p ← all non-constant parameters in p
 7:         if p is empty then
 8:             return misuse = TRUE
 9:         end if
10:         replace each item in p with the prior assignment of item in data flow
11:         increment procedure_counter by 1
12:     end while
13: end for
14: return misuse = FALSE
```

The result of our analyzer is reported in Table 3.2. We applied our checker to 686 (943) applications on GibHub 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.

| **Service** | **Google Cloud** Text2Speech | **AWS** Text2Speech |
|---|---|---|
| Detected constant inputs according to Algorithm 2 | 202 | 196 |
| Confirmed constant inputs inside the main program | 133 | 146 |
| Total | 686 | 943 |

Table 3.2: Number of applications reported as containing constant inputs by Algorithm 2 and those where the constant inputs is in the main program

# CHAPTER 4

# Q: HOW TO EXPOSE FUNCTIONALITY-RELATED PROBLEMS? A: A NOVEL TESTING ALGORITHM - KEEPER

In this chapter, this thesis will discuss certain solutions to functionality-related problems identified in 2.4 in the form of a novel testing solution named Keeper.

## 4.1 Motivation & Discussion of the Relevant Challenges

Recall the *Calling the wrong API* functionality-related anti-pattern discussed in Section 2.4, where the root cause is due to confusion on the expected number of labels to return. However, another issue naturally surrounds the labels themselves. This is highlighted in the example **Phoenix** [8] presented in Section 1.4.

Recall that (as shown in Figure 1.1(b)) Phoenix uses the Google `label_detection` API to perform image classification on an input photo, and then triggers an alarm if any of the top-3 classification labels returned by the API includes the keyword “fire”.

This simple demo application turns out to be difficult to test. First, random inputs work poorly, as they rarely contain fire and hence cannot exercise the critical `alarm()` branch. Second, even with carefully collected image inputs, manual checking is likely needed to judge the execution correctness (i.e., whether an alarm should be triggered). Finally, even after a failed test run—e.g., the picture on the bottom-right corner of Figure 1.1(b) fails to trigger the alarm—it is difficult to know whether the failure is due to the statistical nature of `label_detection`, which has to be tolerated, or the application’s incorrect use of the API, which has to be fixed. In fact, this case belongs to the latter: the bottom-right figure actually has a top-3 label “flame” returned by `label_detection`; not checking for the “flame” label, this application may miss fire alarms in many critical situations.

This example demonstrates several open challenges in testing ML software:

1. **Infinite, yet sparse input spaces**. The spaces of images, texts, or audios —typical input forms of cognitive ML APIs—are infinitely large, yet *realistic* inputs that are *relevant* to the software-under-test are spread sparsely throughout this space. For example, only a tiny portion of real-world images contain fire and are relevant to the fire alarm software. Existing input generation techniques are ineffective here. Random input generators cannot produce realistic inputs through random-pixel images or random-character strings. For example, no fuzzing can turn the left photo into the right photo in Figure 1.1(b).Symbolic execution techniques also do not work, as it is difficult to express the input realism as a solvable constraint. Furthermore, none of these techniques solves the relevance challenge. To tell which images are relevant for a fire alarm application requires both an understanding of the software structure (i.e., knowing that a branch predicate is about fire in the input) and the ability to perform the very cognitive task we need to test (i.e., judging whether a photo contains fire).

2. **Output correctness relying on human judgement**. Cognitive ML APIs are designed to statistically mimic human behaviors, e.g., identifying the objects in an image, interpreting the emotional sentiment in a sentence, etc. Consequently, to judge the correctness of ML software, ideally, we want to ask many people to process the same set of inputs and see if their decisions statistically match with the software outputs—a process that is inherently difficult to automate. For example, it is difficult to tell whether the fire alarm should be triggered or not without manual inspection (Figure 1.1(b)). In traditional testing, the execution correctness often can be checked automatically using the mathematical relationship between the inputs and the outputs or certain invariants expected to hold by the execution. These techniques are still useful for the non-cognitive parts of the ML software, but cannot help the cognitive parts.

3. **Probabilistic incorrectness that is difficult to diagnose**. When ML software produces outputs that differ from most human beings' judgement, which we refer to as

*an accuracy failure*, developers must attribute this failure to either the ML API or the surrounding software's use of the ML API. This attribution is difficult as ML APIs use statistical models to emulate cognitive tasks, and are expected to produce incorrect outputs from time to time. In other words, developers need to distinguish failures caused by the probabilistic nature of the ML API, which simply must be tolerated as part of using this specific ML API, from a misuse of the API, which represents a bug and must be fixed by the developer. Again, this situation is different from that in traditional software testing, where a test failure like a crash indisputably points out something incorrect with the software that needs to be fixed.

Note that a lot of recent works (as discussed in Section 1.5) have studied how to test and fix neural networks. However, they do *not* consider how the neural network is *used* in the context of an application and do *not* test how well the application using the neural network functions.

## 4.2 Overview of Keeper

We present Keeper, a testing tool designed for software that uses cognitive machine learning APIs. A design overview of Keeper is presented in Figure 4.1.

The test input generation part mainly consists of:

1. **Symbolic Execution & Constraint Solving**: Keeper first uses symbolic execution to figure out what values an ML-API output can take to fulfill branch coverage (e.g., "`fire`" == `labels[0].desc` in Figure 1.1(b)). This part only involves traditional program analysis, so symbolic execution & constraint solving would be able to a suitable tool. This is discussed in detail in Section 4.3.1.

2. **Identifying ML inputs (using pseduo-inversion functions)**: To tackle the unique input space and output oracle challenges (bullet point 1 and 2 in Section 4.1), Keeper

designs a set of pseudo-inverse functions for cognitive ML APIs[1]. For an API $f$ that maps inputs from domain $\mathbb{I}$ to outputs in domain $\mathbb{O}$, its pseudo-inverse function $f'$ reverses this mapping at the semantic level. We make sure that the mapping by $f'$ has been confirmed by many people to have high accuracy. For example, the Bing image search engine is a pseudo-inverse function of Google's image classification API. This is discussed in detail in Section 4.3.2.

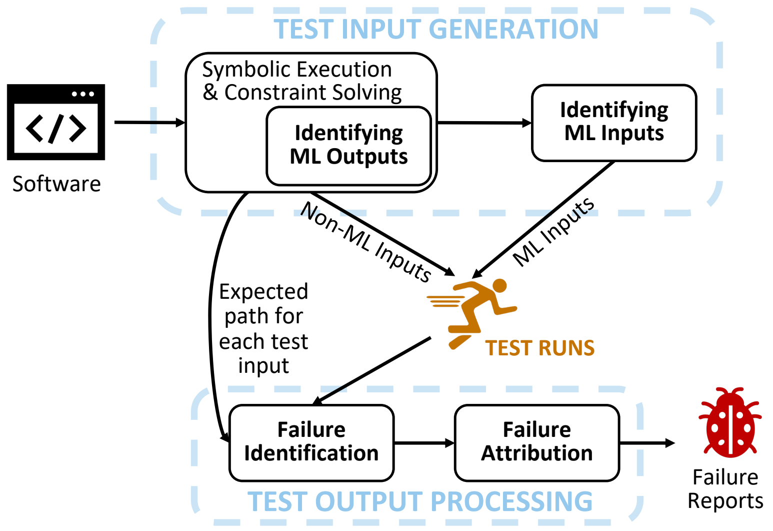


Figure 4.1: An overview of Keeper

Having these two parts at its disposal, Keeper then integrates the pseudo-inverse functions with symbolic execution to reach the sparse program-relevant input space. Specifically, Keeper first uses symbolic execution to figure out the desired ML API outputs. It then automatically generates realistic inputs that are expected to produce the desired ML-API outputs, leveraging pseudo-inverse functions. For example, the two images shown in Figure 1.1(b) are among the images returned by a Bing image search with the keyword "fire".

At this point, Keeper has collected the inputs needed to perform a test run. Keeper also

1. The current implementation of Keeper supports Google Cloud AI APIs and can be easily extended to support similar APIs from other service providers.

makes pseudo-inverse functions a proxy of human judgement and automatically judges the correctness of software outputs that are related to cognitive tasks. Since our pseudo-inverse functions are *not* analytically inverting ML APIs (i.e., $f'(f(i)) \neq i$ is possible), a test input generated by Keeper may not cover the targeted software branch, like the right image in Figure 1.1(b) failing to cover the `alarm` branch. At the same time, since these pseudo-inverse functions have been approved by many human beings, Keeper reports an *accuracy failure* when over a threshold portion of inputs fail to cover a particular target branch. Of course, Keeper also monitors generic failure symptoms like crashes during test runs, and helps expose bugs in code regions that require specific ML inputs to exercise. This part corresponds to the "Failure Identification" part in Figure 4.1 and will be explained in detail in Section 4.4.1.

Finally, to help developers understand the root cause of an accuracy failure, Keeper explores alternative ways of using ML APIs and informs the developers of any code changes that can alleviate the accuracy failure. For the example in Figure 1.1(b), Keeper would inform developers that comparing the returned labels with not only "fire" but also "flame" would make the software behavior more consistent with common human judgement. This part corresponds to the "Failure Attribution" part in Figure 4.1 and will be explained in detail in Section 4.4.2.

## 4.3 Test Input Generation

As outlined in Section 4.2, Keeper decomposes the problem of generating inputs for ML APIs into two parts: first, it identifies the ML API outputs that are needed to satisfy path constraints using symbolic execution (Section 4.3.1); and then synthesizes the ML API inputs that are expected to produce those outputs using carefully designed pseudo-inverse functions (Section 4.3.2). As we will see, this decomposition not only avoids the complexity of directly applying symbolic execution to DNNs, but also help judge the execution correctness (Section

```
def smart_can(img):
  labels = client.label_detection(image=img)
  classes = [x.desc for x in labels]
  for c in classes:
    if  c == "food":
      return "organic"
    if c == "paper" or c == "aluminum":
      return "recyclable"
  return "non-recyclable
```

Figure 4.2: A smart can application, Heap-Sort-Cypher [48]

4.4).

### *4.3.1 ML Output Identification using Constraint Solving*

Keeper's input generation is built upon an existing symbolic execution engine, DSE [47]. Given a function $F$ to test[2] and all the function parameters represented as symbolic variables, a symbolic path constraint is generated for every branch

To identify the desired ML-API outputs, Keeper makes its symbolic execution skip any statement that calls an ML API and instead marks API output that is used by following code as symbolic. This way, the output, instead of input, of ML APIs will be part of the path constraints, and by solving the constraints, Keeper obtains the API output values that are needed to exercise corresponding branches.

This thesis shall refer the readers to Section 3.1 of [2] for a small tweak that Keeper implements related to branch processing.

### *4.3.2 ML Input Identification using Pseduo-inverse Functions*

Given an ML API $f$ and an output $o$, Keeper aims to automatically generate a set of inputs $I$ so that $f(i), i \in I$ is *expected* to produce $o$ according to common human judgement. For example, the two images in Figure 1.1(b) are expected to make `label_detection` output

2. Users of Keeper can choose any function to test, including the `main` function.

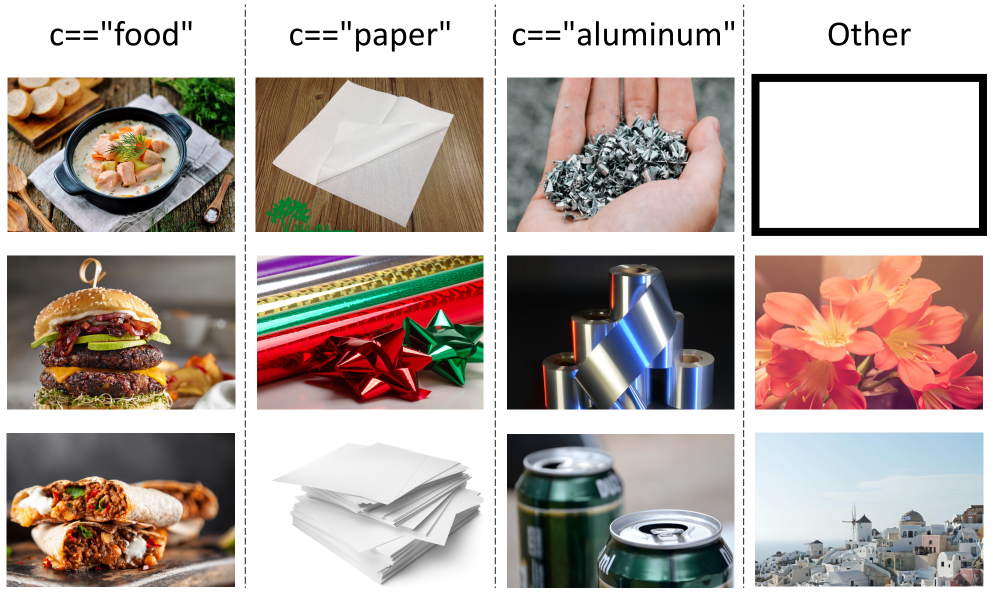


Figure 4.3: Keeper-generated test cases for Figure 4.2

"fire" and the images in every column of Figure 4.3 are expected to make `label_detection` output the corresponding column-header.

To achieve this, Keeper designs a pseudo-inverse function $f'$ for every API $f$, so that $f'(o)$ will produce the input set $I$ for $f$. We want $f'$ to have the following properties.

First, $f'$ is not an analytical inversion of $f$. Ideally, $f'$ should be built independently from $f$ (e.g., not based on the same training data set), so that $f'$ can help not only input generation but also failure identification in a way similar to N-version programming.

Second, $f'$ should be a semantic inverse of $f$, reversing the cognitive task performed by $f$ in a way that is consistent with most human beings. This way, test inputs generated by Keeper can expect to cover most of the software branches, unless the ML API is unsuitable for the software or is used incorrectly.

Third, $f'$ should produce more than one output for each input it takes in. This will allow Keeper to generate multiple inputs for $f$ to exercise a corresponding branch, and get a statistically meaningful test result given the probabilistic nature of ML APIs.

With these goals in mind, we design a total of 8 pseudo-inverse functions as summarized in Table 4.1.

We characterize these 8 API inverses into the following three categories.

| ML Category | Vision | |
|---|---|---|
| ML Task | Image classification | Object detection |
| ML API Output | image class | object name |
| Constraint Example | `class == "fire"` [8] | `object == "tableware"` [49] |
| Pseudo-inverse Function | Search on internet w. keyword: [image class] | Search on internet w. keyword: [object name] |
| ML Category | Vision | |
| ML Task | Face detection | Text detection |
| ML API Output | face emotion | extracted text |
| Constraint Example | `emotion == "joy"` [50] | `text == "3923-6625"` [51] |
| Pseudo-inverse Function | Search on internet w. keyword: [emotion] + "human face" | Print [extracted text] on an image |
| ML Category | Speech | Language |
| ML Task | Speech recognition | Entity detection |
| ML API Output | transcript | entity name, type |
| Constraint Example | `text == "turn on the light"` [52] | `type == "PERSON"` [53] |
| Pseudo-inverse Function | Use speech synthesize technique on [transcript] | Use text generation technique w. seed: [name] or [type] |
| ML Category | Language | |
| ML Task | Document classification | Sentiment detection |
| ML API Output | document class | score, magnitude |
| Constraint Example | `class == "food"` [54] | `score < 0` [55] |
| Pseudo-inverse Function | Search on internet w. keyword: [document class] | Select tweets from Sentiment140 dataset [56] |

Table 4.1: Different ML APIs handled by Keeper with pseudo-inverse functions

## Search-based Pseudo-inverses

For many vision and language APIs, search engines offer effective pseudo inversion: they take in a keyword and return a set of realistic images/texts that reflect the keyword. Search engines have several properties that serve Keeper's testing purposes. First, they offer great semantic inversion, as there are multiple search engines that have been used by hundreds of millions of users for many years with high satisfaction. Their top search results typically match the common human judgement. Second, they are not an analytical inversion of ML

APIs, and we will use non-Google engines to minimize potential correlations. Third, they accept a wide range of search words and produce many ranked results, which means a large number of high-quality test inputs for Keeper. Specifically, Keeper uses different engines and search keywords for different ML APIs:

1-2. **Image classification** and **Object detection** APIs return string labels that describe the image and the objects inside the image, respectively. For both APIs, Keeper uses the Bing [57] image search engine and uses the desired label description or object name as the search keyword. For example, the images in each column of Figure 4.3 were the top-3 search results returned by Bing using the keywords listed atop. The only exception is the last column: when there is no specific keyword requirement (like `c != food and c!= paper and c != aluminum`), Keeper uses a blank image and images generated by a random-image generator [58].

3. The **Face detection** API detects human faces in an image. Some ML software uses the returned emotion string associated with each face (e.g., "joy", "sorrow", etc.) to decide execution path. To generate corresponding images, Keeper uses "[emotion] human face" as a keyword to search the Bing image.

4. **Document classification** APIs process a document and return categories based on the document content, like "pets", "health", "sports", and others. Keeper uses the desired category name as keyword and searches it at (1) knowledge graph websites, Wikipedia [59] and Britannica [60]; and (2) Bing web search engines. Keeper then uses the text extracted out from each returned web page as the ML API input.

### Synthesis-based Pseudo-inverses

The semantic inversion of some ML APIs does not match the functionality of search engines. Fortunately, we find ways to synthesize inputs for them.

5. The **Text detection** API extracts printed or handwritten text from an image. Unfortunately, image search engines tend to return images whose content reflects the search keyword, instead of images that contain the keyword as text within the image. Therefore, given a text string, Keeper prints it on a background image using the Python pillow library [61]. Keeper adopts both printed and hand-writing fonts; different font settings produce different test images. To decide the background image, Keeper checks whether the `text-detection` API shares its input image with another vision API. If so, the test images Keeper generated for the other API will be used as the background; otherwise, a blank image and some random images will be used. Figure 4.4 shows some of the test images that Keeper generates for application wanderStub [62], which has a branch checking if the input image contains "Total".

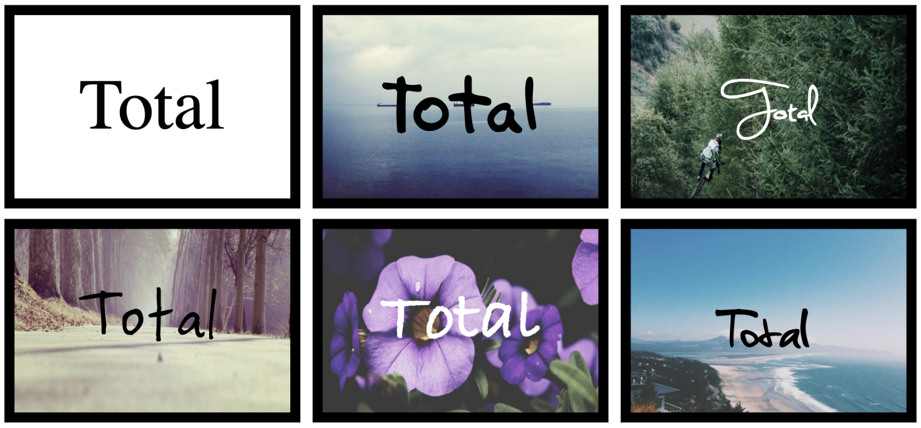


Figure 4.4: Test inputs generated for wanderStub [62].

6. The **Entity detection** API inspects the input sentence for known entities—there are in total 13 entities, such as ADDRESS, DATE, etc. Since the search engines usually return long documents, Keeper instead uses a popular language model GPT-2 [63] to synthesize any number of sentences that start with a pre-defined word/phrase that corresponds to the desired entity type.

   Specifically, the Entity detection API output entity names and types. For entity names, Keeper uses them as cues to solicit responses from GPT-2. For entity types, Keeper sets up a 13-entry table to map each entity type into a list of concrete words or phrases. Then, given an entity type, Keeper feeds corresponding words/phrases into the GPT-2 model, which then generates any specified number of sentences that start with that word/phrase. For instance, for the LOCATION type, one cue (among others) that Keeper uses is "Chicago is", and one corresponding GPT-2 output could be:

> Chicago is growing at a slow rate. In November 2014, the report noted that sales were down slightly as they focused on business in neighborhoods. "Nearly half the places from the new high-rise SRT," it said, were being moved to 10.

7. The **Speech recognition** API transcribes the input audio clip and outputs the transcript. Keeper uses speech synthesis tools, particularly the pyttsx3 [64] Python library, to generate the desired audio clips based on a given transcript. Keeper generates multiple audio clips using different voice settings supported by this library.

## ML-benchmarks-based Pseudo-inverses

8. The **Sentiment detection** API presents two challenges. First, although this API aims to identify the prevailing emotional opinion within the text, it does not directly output a categorical result. Instead, it returns two floating-point numbers, `score` and `magnitude`, for developers to derive emotion categories from. There is no perceivable way to generate text that can offer the exact `score` or `magnitude`. Second, even if we just hope to generate text that contains positive or negative emotion, no search engine or synthesizer can accomplish this.

   Facing these challenges, Keeper resorts to the Sentiment140 dataset [56], which contains 1,600,000 tweets, manually labelled as positive, negative, and neutral. Keeper randomly samples the same number of positive, negative, and neutral tweets as test inputs for any sentiment-detection API called inside an ML software, with the expectation that these tweets will help cover different branches in the software that are designed for different emotions.

Note that, we treat ML benchmarks as the last resort for multiple reasons:

- First, the labels associated with data inside ML benchmarks either have few categories or have limited quality. For example, ImageNet [65] contains 1000 manually labeled image categories, which is too few compared with the 20,000 labels of Google Vision AI. On the contrary, OpenImage has 9 million images with 20,000 labels. However, 89% of the labels are generated by DNNs, and 53% of the human-verified ones are incorrect [66].
- Second, ML benchmarks are built with pre-processed real-world data. Such ”clean” data has less variety, as they share similar size, resolution, and encoding format.
- Third, some benchmarks may be part of the training data set of Google ML APIs, which makes the test inputs biased towards the ones APIs can perform well on and hence less likely to reveal problems.
- Finally, Generative Adversarial Network synthesizes new data following the distribution of the training set [67]. We do not use it, as this approach requires much training data and ends up generating non-real-world data that has similar distribution with the training set, whose limitations we discussed earlier.

## 4.4 Test Output Processing

Once relevant test inputs are collected, Keeper runs the test and processes the output.

### *4.4.1 Failure Identification*

Keeper looks for three types of failure symptoms: (1) low accuracy, (2) dead code, and (3) generic failures like crashes.

1. **Low accuracy failures**: For all the inputs $\mathbb{I}_b$ that are generated to cover a branch $b$, Keeper checks which of them exercise $b$ at run time, denoted as $\mathbb{I}_b^{\text{succ}}$ and calculates the *recall* of $b$ (i.e., $\frac{|\mathbb{I}_b^{\text{succ}}|}{|\mathbb{I}_b|}$). If the recall drops below a threshold $\alpha$, 75% by default. Keeper reports an accuracy failure associated with $b$. The setting of $\alpha$ can be adjusted, but should not be 100%, as ML APIs are probabilistic and pseudo-inverse functions cannot guarantee to be correct all the time.

   Note that, these accuracy failures are **not** equivalent with low precision or low recall of the ML API itself. The latter is just one of the possible root causes of the former. keeper intentionally does not calculate the precision or recall of any ML API, but instead focuses on the overall software.

2. **Dead code failures**: These occur when a branch is not covered after all the testing runs.

3. **Generic failures** These have symptoms like crashes that do not require special techniques to observe.

This thesis shall refer readers to Section 4.1 of [2] for a detailed discussions on the failures mentioned above.

### *4.4.2 Failure Attribution*

To help developers understand and tackle accuracy failures, Keeper attempts to automatically patch the software by changing how ML APIs' output is used. Keeper suggests the change to developers and if all attempts failed, Keeper suggests developers to consider using a different, more accurate ML API, or adding extra input screening or pre-processing. Specifically, Keeper attempts two types of changes to the branch $b$ where the failure is associated with:

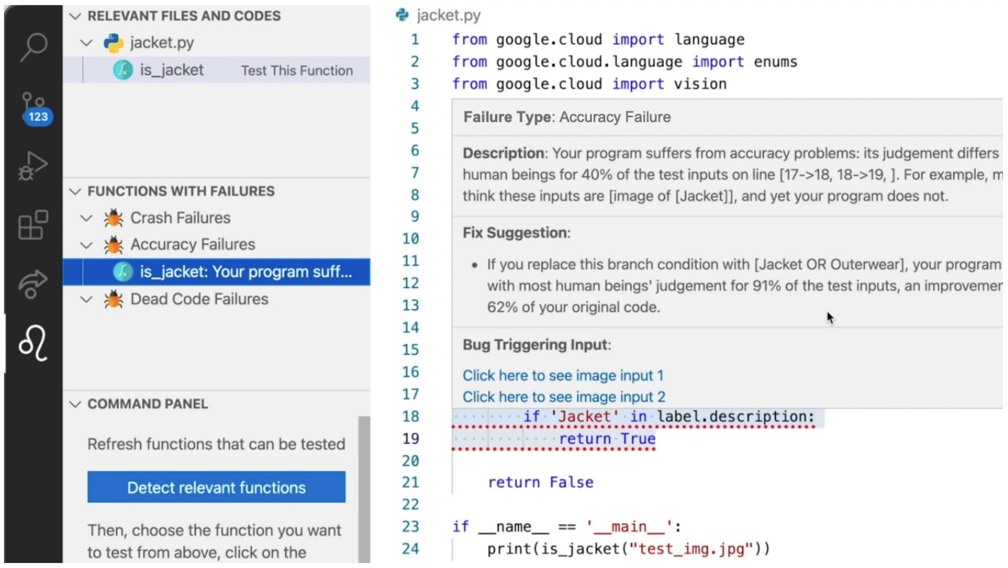
1. **Label changes**: When branch $b$ compares a ML API output with a set of labels, (e.g. the branches in Figure 1.1(b) and 4.2), Keeper tries to expand the set of labels with three goals in mind. (1) Recall goal: more test inputs that are expected to exercise $b$ can now satisfy $b$'s condition; (2) Precision goal: most inputs that are not expected to exercise $b$ should continue to fail the condition of $b$; (3) Semantic goal: the added labels are related to the original label(s) in $b$ in terms of natural language semantics. For the exact algorithm, see Section 4.2 of [2].

2. **Threshold changes**: See Section 4.2 of [2] for details.

## 4.5 Implementation & VS Code IDE Plugin

We have implemented Keeper for Python applications that use Google Cloud AI APIs [3], the most popular cloud AI services on Github [1]. The core algorithm of Keeper is general to other languages and ML Cloud APIs. Keeper uses dynamic symbolic execution framework PyExZ3 [47], which implements the DSE algorithm, and uses CVC4 [68] for constraint solving. Keeper uses Python built-in trace back tool [69] to check branch coverage, and Pyan [70] and Jedi [71] for call graph and program dependency analysis. Keeper uses Python scikit-learn [72] library for linear regression models (for why this is needed, see Section 4.2 of [2]).

We have implemented an IDE plugin for visualized interaction with Keeper, as the debugging and fixing of accuracy failures particularly requires developers' participation, as illustrated in Figure 4.5. The plugin is an extension in Visual Studio Code [73], a popular code editor supporting multiple languages. For any Python software, Keeper first identifies



Refresh functions that can be tested

Then, choose the function you want to test from above, click on the

```
from google.cloud import language
from google.cloud.language import enums
from google.cloud import vision
```

**Failure Type**: Accuracy Failure

**Description**: Your program suffers from accuracy problems: its judgement differs human beings for 40% of the test inputs on line [17->18, 18->19, ]. For example, m think these inputs are [image of [Jacket]], and yet your program does not.

**Fix Suggestion**:

- If you replace this branch condition with [Jacket OR Outerwear], your program with most human beings' judgement for 91% of the test inputs, an improvemer 62% of your original code.

**Bug Triggering Input**:

Click here to see image input 1
Click here to see image input 2

```
        if 'Jacket' in label.description:
            return True

    return False

if __name__ == '__main__':
    print(is_jacket("test_img.jpg"))
```

Figure 4.5: Keeper IDE plugin interface

all functions that invoke ML APIs directly or indirectly through callees, and displays them on the side bar, under “RELEVANT FILES AND CODES” in Figure 4.5. From that list, developers can select the function to test [3]. Once they have made the selection, they will be asked to provide type information of function parameters, as Python is a dynamically typed language. Keeper will then start testing as outlined in Section 4.2. At the end of the testing, which usually takes 1–2 minutes, any execution failure that has been exposed is listed under the “FUNCTIONS WITH FAILURES” sidebar, as shown in the figure. Source code related to each failure is highlighted, together with a hovering window that offers detailed information like failure description, triggering inputs, and patch suggestions. Developers can subsequently clear the highlights to make modifications using buttons under the “COMMAND PANEL” sidebar [4]. The IDE plugin is published in VS Code Marketplace, featuring an easy installing process integrated in the VS Code ecosystem [74] [5].

3. Users can also specify other function (and the file it is contained in) not displayed here to test through an IDE command.

4. and display the highlights again, if needed.

5. Tutorials and examples are available on that site as well.

## 4.6 Evaluation

### *4.6.1 Software Testing Evaluations*

We evaluate Keeper using 63 Python applications that are from two sources:

1. From the 360 open-source applications assembled by a previous study of ML APIs [1] , we found 45 Python applications that use ML APIs in a non-trivial way (i.e., the API output affects control flow).

2. We additionally checked about 100 random Python applications on GitHub that use ML APIs and found 18 applications that use ML APIs in a non-trivial way.

For more information on these applications, please see Section 6.1.1 of [2].

We select three baselines for comparison (Section 6.1.2 of [2]).

Evaluated on branch coverage, across different types of applications, Keeper consistently performs well, around 90% on average. The uncovered branches are either related to dead-code failures that Keeper discovers, or related to code that our underlying symbolic execution engine cannot handle. Keeper's branch coverage is consistently better than other baselines (see Section 6.2.1 of [2] for details).

In terms of locating bugs, Keeper exposed many failures: 35 failures from the latest version of 25 applications. These failures cover a range of symptoms and root causes. Except for one failure caused by missing type conversion, the others are all related to different types of cognitive ML tasks. In comparison, alternative testing techniques missed 2–3 crash failures caught by Keeper. Furthermore, unlike Keeper, they cannot automatically recognize accuracy failures and dead-code failures.

For details, see Section 6.2.2 of [2].

### *4.6.2 User Studies*

To better evaluate the accuracy failures and the code changes suggested by Keeper, we recruited 100 participants on Amazon Mechanical Turk (Mturk) for a software-user survey. The survey includes 4 applications from our benchmark suites: 2 image-related applications and 2 text-related applications. On each survey page, a brief description is given for an application and user-study participants are told to review how two versions of this application perform on a set of inputs. Then, the web page displays a number of input images/text and the corresponding outputs of application version-1 and application version-2. These two versions are the original application and the application with suggested code changes from Keeper (referred to as *fixed* in Figure 4.6); we randomly decide which one of them is version-1 and which is version-2 on each survey page to reduce potential bias. Each participant is asked to answer questions about (1) for each input, which version's output they prefer; and (2) which version they think is better with everything considered. Participants were compensated $5 after the survey.

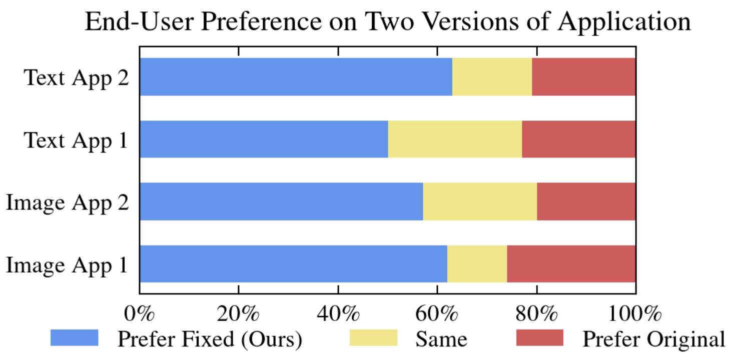


Figure 4.6: End-user preference: Original vs. Keeper version.

A summary of the user study results is shown in Figure 4.6. As we can see, in all cases, a dominate portion of end-users prefer the version with changes suggested by Keeper over the original version, supporting Keeper's judgement about accuracy failures and Keeper's attempt in fixing the accuracy problems. At the same time, we also noticed that there are 20–26% of user-study participants who prefer the original software and 12–27% who feel the two versions are about the same. These results confirm the fact that cognitive tasks are inherently subjective—even human beings often do not agree with each other on these tasks.

In addition, we recruited 10 participants who have Python programming experience. Half of them are software engineers from industry and half are college students. Given ML API official documents, they are asked to implement two functions, one for image analysis and one for text analysis, with the code skeleton provided by us. They then use Keeper to test their implementation. For any failures reported by Keeper, they are asked about whether they think the failure indeed reflects a software bug; how they would fix the code; whether they think Keeper is helpful; and others. This whole process is conducted through Zoom, with two researchers remotely interacting with the participant. Participants were compensated $10 after the interview. At the end of all interviews, three researchers coded the interview data independently and then met to resolve disagreements.

It took 20–60 minutes for the participants to read ML API documents and program; the whole session took 40–90 minutes.

In total Keeper reported 12 accuracy failures, 3 generic failure, and 6 dead-code failures for the 20 implemented functions from 10 participants. Keeper suggested code changes for all the 12 accuracy failures. Only one participant (P3) managed to program both functions correctly. The other 9 participants each has 1–4 failures exposed by Keeper from their code.

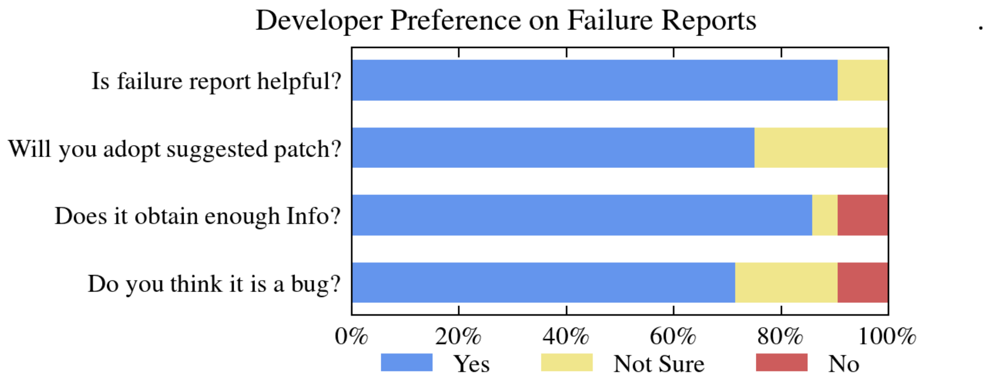


Figure 4.7: Developer preference of Keeper failure reports.

Figure 4.7 and Table 4.2 summarized the developer interview results. As we can see,

| Question | Answer | # |
|---|---|---|
| Do you think Keeper is helpful? | Yes, it is useful. | 9 |
| | It would be helpful if becomes faster. | 1 |
| Do you like the interface of Keeper? | I like it. | 6 |
| | Need guidance to use/read it. | 3 |
| | Hope it could show more info. | 1 |

Table 4.2: Developer overall preference of Keeper.

almost all accuracy-failure reports (18 out of 21) are regarded as helpful. For most of the accuracy failure reports (9 out of 12), participants said they would definitely adopt the patch suggested by Keeper. Participants were not sure about the suggested patch in 3 reports, because they wanted to inspect Keeper-generated test cases before making decisions. Participants strongly agreed that most failure reports (15 out of 21) pointed out bugs in their programs, including 7 out of 12 accuracy failures. There were only two cases, both about accuracy failures, where participants felt the failures do not mean their code is buggy, although in both cases they were willing to adopt the code changes suggested by Keeper. We believe this reflects the subjective nature of cognitive tasks.

Near the end of each user study session, we asked the participants "Do you think Keeper is helpful?". Overwhelmingly, they answered "Yes" (9 out of 10). They told us that "I don't know much about machine learning, but this tool helps a lot" (P1); "I like it tell me how accurate my code is." (P4); "It's cool. I have no idea how it finds these more optimized solutions." (P5); "Hope my team could adopt similar testing tool." (P7); "Showing failure cases help me to troubleshoot." (P8). Many of them like the user interface after learning how to use it with no help from us (6 out of 10). They told us that "The tool is intuitive. I like the little symbols." (P1); "The UI interface is quite clear to use. It is even better than some old industry products." (P3); "I like the sidebar display." (P7).

# CHAPTER 5

# CONCLUSION & DISCUSSION: WHERE TO GO NEXT?

This thesis has made the following major contributions aimed at understanding and improving the usage of ML cloud APIs:

1. A comprehensive empirical study on 360 representative open-source GitHub applications, which generalizes anti-patterns that degrades the functionality, performance, or cost-efficiency of these software;

2. On the performance front, the developments of two static analyzers targeting asynchronous API usage and repetitive API calls, which detects around 500 applications on GitHub as containing these mis-uses;

3. On the functionality front, the proposal, implementation, and evaluation of a novel testing tool (Keeper) that proves successful in tackling the tricky problem of testing ML cloud APIs.

This thesis shall make a few remarks before concluding.

**Application selection**: In this series of research, our benchmark applications are drawn from open-source applications on GitHub. During the application selection process, a significant amount of effort was devoted on filtering out applications whose usages of ML APIs are *trivial* (see Section 2.2). However, even with such efforts, it is still difficult to sample the rest of *real-world* applications using ML APIs, which Github open-source applications are inherently not a part of. This is one threat that this thesis acknowledges. However, the problems and solutions mentioned are likely to generalize, from this thesis's opinion, to other applications using these (or a similar set of) ML APIs, especially regarding the functionality of software in relation to cognitive ML APIs.

**Static analyzer of performance-related static checkers**: The analyzers tackling performance-related mis-uses, as described in Chapter 3, are static. Although it is possible to consider possible other further work (e.g. dynamic analyzers), this thesis believes it is of less importance than tackling other more pressing problems degrading application functionality. However, there are still possible future directions, including but not limited to, integrating these checkers into IDE plugins and conducting HCI-related research on asynchronous API usage.

With that, this thesis concludes this line of research.

## REFERENCES

[1] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "Are Machine Learning Cloud APIs Used Correctly?," in *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, p. 125–137, IEEE Press, 2021.

[2] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Automated Testing of Software that Uses Machine Learning APIs," in *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*, 2022.

[3] Google, "Google Cloud AI." Online document, 2022.

[4] Amazon, "Amazon artificial intelligence service." Online document, 2022.

[5] IBM, "IBM Watson." Online document, 2022.

[6] Microsoft, "Microsoft Azure Cognitive Services." Online document, 2022.

[7] WhatsInYourFridge, "A smart fridge application." Github application, 2021.

[8] Phoenix, "A fire-detection application." Github application, 2022.

[9] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Testing Deep Neural Networks." arXiv pre-print, 2018.

[10] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, (New York, NY, USA), p. 109–119, Association for Computing Machinery, 2018.

[11] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), p. 1–18, Association for Computing Machinery, 2017.

[12] 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 *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '19, (New York, NY, USA), p. 146–157, Association for Computing Machinery, 2019.

[13] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Structural test coverage criteria for deep neural networks," *ACM Trans. Embed. Comput. Syst.*, vol. 18, October 2019.

[14] C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, and M. J. Kochenderfer, "Algorithms for verifying deep neural networks," *Foundations and Trends in Optimization*, vol. 4, no. 3-4, pp. 244–404, 2021.

[15] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Deepconcolic: Testing and debugging deep neural networks," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, p. 111–114, IEEE Press, 2019.

[16] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), p. 303–314, Association for Computing Machinery, 2018.

[17] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), p. 1016–1026, Association for Computing Machinery, 2018.

[18] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, p. 1158–1161, IEEE Press, 2019.

[19] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. P. J. C. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '18, (New York, NY, USA), p. 118–128, Association for Computing Machinery, 2018.

[20] 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 *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '20, (New York, NY, USA), p. 851–862, Association for Computing Machinery, 2020.

[21] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '20, (New York, NY, USA), p. 211–224, Association for Computing Machinery, 2020.

[22] S. Lee, S. Cha, D. Lee, and H. Oh, "Effective white-box testing of deep neural networks with adaptive neuron-selection strategy," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '20, (New York, NY, USA), p. 165–176, Association for Computing Machinery, 2020.

[23] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. Vanderplas, A. Joly, B. Holt, and G. Varoquaux, "Api design for machine learning software: experiences from the scikit-learn project." arXiv pre-print, 2013.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library." arXiv pre-print, 2019.

[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, (USA), p. 265–283, USENIX Association, 2016.

[26] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries," in *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, p. 1027–1038, IEEE Press, 2019.

[27] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, (New York, NY, USA), p. 486–498, Association for Computing Machinery, 2020.

[28] Amazon, "AWS Lambda." Online document, 2022.

[29] Google, "Speech-to-Text API Basics." Online document, 2022.

[30] Amazon, "Comprehend API Document Processing Modes." Online document, 2022.

[31] AnsweringMachine, "A bot application." Github application.

[32] S. Kakar, "How to Create a Sales Cadence That Doesn't Annoy Prospects." Online document, 2019.

[33] JiangJungDian, "A meeting management application." Github application.

[34] Google, "Speech-to-Text Perform streaming speech recognition on a local file." Online document, 2022.

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

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

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

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

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

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

[41] 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.

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

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

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

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

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

[47] M. Irlbeck *et al.*, "Deconstructing dynamic symbolic execution," *Dependable Software Systems Engineering*, vol. 40, p. 26, 2015.

[48] HeapSortCypher, "A garbage classification application." https://github.com/matthew-chu/heapsortcypher.

[49] recipeGo, "A recipe recommendation application." https://github.com/Reckonzz/recipeGO.

[50] emotion2music, "A smart music player application." https://github.com/varnachandar/emotion2music.

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

[52] TRANSLATOR, "A smart light application." https://github.com/mubeenafatima/TRANSLATOR.

[53] Klassroom, "A lecture note application." https://github.com/dev5151/Klassroom.

[54] noteScript, "A lecture note application." https://github.com/GalenWong/noteScript.

[55] stockmine, "A stock prediction application." https://github.com/nicholasadamou/stockmine.

[56] A. Go, R. Bhayani, and L. Huang, "Twitter sentiment classification using distant supervision," *CS224N project report, Stanford*, vol. 1, no. 12, p. 2009, 2009.

[57] M. Bing, "Bing image search." https://www.bing.com/images/trending?FORM=ILPTRD.

[58] "Lorem picsum." https://picsum.photos.

[59] "Wikipedia." https://en.m.wikipedia.org/.

[60] "Encyclopedia britannica." https://www.britannica.com/.

[61] "Pillow: Python imaging library." https://pypi.org/project/Pillow/.

[62] WanderStub, "An exchange conversion application." https://github.com/richardjpark26/WanderStub.

[63] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[64] "pyttsx3: Text-to-speech library for python." https://pypi.org/project/pyttsx3/.

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

[66] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, *et al.*, "The open images dataset v4," *International Journal of Computer Vision*, pp. 1–26, 2020.

[67] J. Gui, Z. Sun, Y. Wen, D. Tao, and J. Ye, "A review on generative adversarial networks: Algorithms, theory, and applications," *arXiv preprint arXiv:2001.06937*, 2020.

[68] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 171–177, Springer, 2011.

[69] "Python system-specific parameters and functions." https://docs.python.org/3/library/sys.html#sys.settrace.

[70] D. Marby and N. Yonskai, "Pyan3: Offline call graph generator for python 3." https://github.com/davidfraser/pyan.

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

[72] "scikit-learn: Machine learning in python." https://scikit-learn.org/stable/.

[73] Microsoft, "Visual studio code." Online document https://code.visualstudio.com/, 2021.

[74] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Keeper vs code ide plugin." Online document https://marketplace.visualstudio.com/items?itemName=ALERTProject.mlapitesting, 2022.