ChameleonAPI: Automatic and Efficient Customization of Neural Networks for ML Applications

Yuhan Liu, *University of Chicago;* Chengcheng Wan, *East China Normal University;* Kuntai Du, Henry Hoffmann, and Junchen Jiang, *University of Chicago;* Shan Lu, *University of Chicago and Microsoft Research;* Michael Maire, *University of Chicago*

https://www.usenix.org/conference/osdi24/presentation/liu

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

ChameleonAPI: Automatic and Efficient Customization of Neural Networks for ML Applications

Yuhan Liu[‡], Chengcheng Wan[*], Kuntai Du[‡], Henry Hoffmann[‡], Junchen Jiang[‡], Shan Lu[‡†], Michael Maire[‡]
[‡]*University of Chicago* [*]*East China Normal University* [†]*Microsoft Research*

Abstract

ML APIs have greatly relieved application developers of the burden to design and train their own neural network models—classifying objects in an image can now be as simple as one line of Python code to call an API. However, these APIs offer the same pre-trained models *regardless* of how their output is used by different applications. This can be suboptimal as not all ML inference errors can cause application failures, and the distinction between inference errors that can or cannot cause failures varies greatly across applications.

To tackle this problem, we first study 77 real-world applications, which collectively use six ML APIs from two providers, to reveal common patterns of *how ML API output affects applications' decision processes*. Inspired by the findings, we propose *ChameleonAPI*, an optimization framework for ML APIs, which takes effect without changing the application source code. ChameleonAPI provides application developers with a parser that automatically analyzes the application to produce an abstract of its decision process, which is then used to devise an application-specific loss function that only penalizes API output errors critical to the application. ChameleonAPI uses the loss function to efficiently train a neural network model customized for each application and deploys it to serve API invocations from the respective application via existing interface. Compared to a baseline that selects the best-of-all commercial ML API, we show that ChameleonAPI reduces incorrect application decisions by 43%.

1 Introduction

The landscape of ML applications has greatly changed, with the rise of ML APIs significantly lowering the barrier of ML application developers. Instead of designing and managing neural network models by themselves via frameworks like TensorFlow and PyTorch, application developers can now simply invoke ML APIs, provided by open-source libraries or commercial cloud service providers, to accomplish common ML tasks like object detection, facial emotion analysis, etc. This convenience thus gives rise to a variety of ML applications on smartphones, tablets, sensors, and personal assistants [9, 29, 50, 65].

Although ML APIs have eased the integration of ML tasks with applications, they are suboptimal by serving different applications with the same neural network models. This issue is particularly striking when applications use the ML API results to make control-flow decisions (also referred to as *application decisions* in this paper). Different applications may check the result of the same ML API using different control-flow code structures and different condition predicates, a process that we refer to as the application's *decision process* (see §2 for the formal definition). Due to the heterogeneity across applications' decision processes, we make two observations.

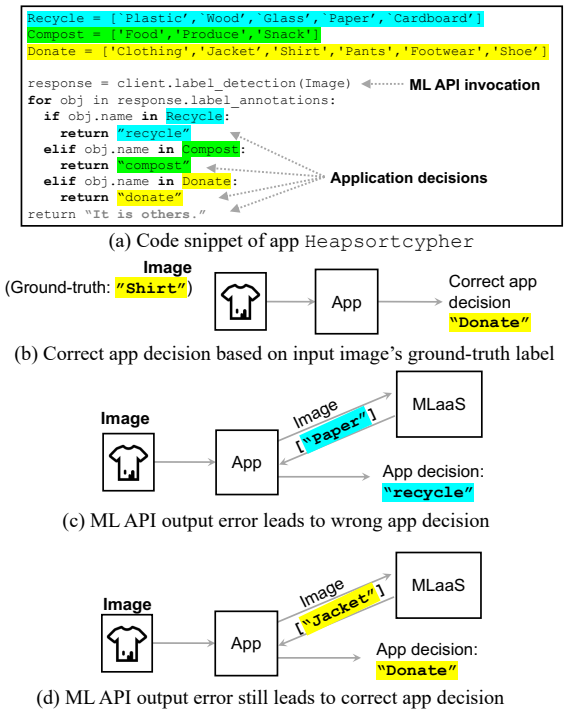
- First, some incorrect ML API outputs may still lead to correct application decisions, with only certain *critical errors* of API output affecting the application's decision.
- Second, among all possible output errors of an ML API, which ones are critical *vary* significantly across applications that use this API. That is, the same API output error may have a much *greater* effect on one application than on another.

Figure 1 illustrates the decision process of a garbage-classification application `Heapsortcypher` [49]. It first invokes Google's classification API upon a garbage image. Then, based on the returned labels, a simple logic is used to make the *application decision* about which one of the pre-defined categories (`Recycle`, `Compost`, and `Donate`) or `others` the image belongs to. For example, for an input image whose ground-truth label is `"Shirt"`, the correct application decision is `Donate`, as shown in Figure 1 (b).

For this application, when the classification API fails to return `"Shirt"`, the application decision may or may not be wrong. For example, Figure 1 (c) and (d) show two possible wrong API output: if the output is `"Paper"`, the application will make a wrong decision of `Recycle`; however, if the output is `"Jacket"`, the application will make the correct decision of `Donate` despite not matching the ground-truth label. More subtly, if the API returns a list of two labels, `"Shirt"` and `"Paper"`, the application would make a correct decision if `"Shirt"` is ordered before `"Paper"` by the API, but would make a wrong decision if `"Paper"` is ordered before `"Shirt"`. The reason is that the application logic, the `for` loop in Figure 1 (a), checks one API-output label at a time. As we will see later, there are also other ways that applications check the API-output list, which will affect application decision differently.

As we can see, for a specific application, some errors of an ML API may be critical, like mis-classifying the shirt to `"Paper"` in the example above, and yet some errors may be non-critical, like mis-classifying the shirt image as `"Jacket"` or classifying the shirt image as both `"Shirt"` and `"Paper"` in the examples above. Which errors are critical varies, depending on the application's decision process.

These observations regarding the critical errors specific to



(a) Code snippet of app `Heapsortcypher`

(b) Correct app decision based on input image's ground-truth label

(c) ML API output error leads to wrong app decision

(d) ML API output error still leads to correct app decision

Figure 1: *An example ML application whose decision depends on the output of ML API (multi-label classification), but not all errors of ML API output have the same effect.*

each application suggest substantial room for improvement by customizing the ML API, essentially the neural network model underneath the API, for individual application's decision process. In particular, for a given application, the customized model can afford having more errors less critical to the application for the benefit of having fewer critical errors that cause wrong application decisions.

Thus, our goal is to allow ML APIs and their underlying neural network models to be *automatically* customized for a given application, so as to *minimize* incorrect application decisions *without* changing the application's source code or interface between ML API and software exposed to developers. This way, application developers who do not have the expertise to design and train customized ML models can still enjoy the accessibility of generic ML APIs while getting closer to the accuracy of ML models customized for the application.

No prior work shares the same goal as us. The closest line of prior work specializes DNN models for given queries [7, 8, 36, 37, 43], but they require application developers to use a domain specific language (*e.g.,* in SQL [36]) instead of general programming languages, like Java and Python, and mostly focus on reducing the DNN's size. In contrast, we keep both the ML API interface and the application source code intact while avoiding incorrect decisions for ML applications.

With the aforementioned goal, this paper makes two contributions. *First,* we run an empirical study over 77 real-world applications that collectively use six ML APIs to reveal several common patterns of how the outputs of ML APIs affect the application decisions (§2).

Our study identifies two types of ML API output that are used by applications to make control-flow decisions (categorical labels and sentiment scores), and three types of decision types (True-False, Multi-Choice, and Multi-Selection) with different implications regarding which ML API output errors are critical to the application.

Our study also quantitatively reveals opportunities of model customization. (1) Although popular image-classification models are trained to recognize as many as 19.8K different labels, the largest number used by any one application for decision making is only 54. Consequently, mis-classification among the remaining tens of thousands of labels are completely irrelevant to an application. (2) More importantly, applications tend to treat multiple labels (4.7 on average) as one equivalence class in their decision making, such as labels `Plastic`, `Wood`, `Glass`, `Paper`, and `Cardboard` in Figure 1(a). Mis-classification among those labels inside one equivalence class does not matter. (3) Which labels are relevant to an application's decision making vary greatly across applications, with only 12% of application pairs share any labels used for their decision making.

Second, inspired by the empirical study, we propose ChameleonAPI, which customizes and serves ML models behind the ML API for each given application's decision process, without any change to the existing ML API or the application source code (§3). ChameleonAPI works in three steps. First, it provides a parser that analyzes application source code to extract information about how ML inference results are used in the application's decision process. Based on the analysis result, ChameleonAPI then constructs the loss function to reflect which ML model output is more relevant to the given application as well as the different severity of ML inference errors on the application decisions. The ML model will be retrained accordingly using the new loss function. Finally, when the ML API is invoked by the application at runtime, a customized ML model will be used to serve this query.

We evaluate ChameleonAPI on 57 real-world open-source applications that use Google and Amazon's vision and language APIs. We show that ChameleonAPI's re-trained models reduce 48% of incorrect decisions compared to the off-the-shelf ML models and 50% compared to the commercial ML APIs. Even compared with a baseline that selects the best-of-all commercial ML API, ChameleonAPI reduces 43% of incorrect decisions. ChameleonAPI only takes up to 24 minutes on a GeForce RTX 3080 GPU to re-train the ML model.

Our code is publicly available at `https://github.com/UChi-JCL/chameleonAPI`.

2 Understanding Application Decision Process

We conduct an empirical study to understand how applications make decisions based on ML APIs (§2.3), and how this

ML API name	ML task	Provider	# of apps
label_detection	Vision::Image classification	Google	29
detect_labels	Vision::Image classification	Amazon	11
object_localization	Vision::Object detection	Google	8
analyze_sentiment	Language::Sentiment analysis	Google	14
analyze_entities	Language::Entity recognition	Google	6
classify_text	Language::Text classification	Google	9

Table 1: *Summary of applications used in our empirical study.*

decision making logic implies the different severity of ML inference errors (§2.4). This study will reveal why and how to customize the ML API backend for each application. As a representative sample of ML APIs, this study focuses on cloud AI services due to their popularity.

2.1 Definitions

Preliminaries: We begin with basic definitions.

- *Application decision*: the collective control-flow decisions (i.e., which branch(es) are taken) made by the application under the influence of a particular ML API output.
- *Incorrect ML API output:* a situation when the API output differs from the API input's human-labeled ground truth. We refer to such ML API outputs as *API output errors*.
- *Correct decision*: the application decision if the API output is the same as the human-labeled ground-truth of the input.
- *Application decision failure*: a situation when the application decision is different from the correct decision, also referred to as *application failure* for short in this paper.

Software decision process: Given these definitions, an application's *software decision process* (or decision process for short) is the logic that maps an ML API output to an application decision. The code snippet in Figure 1 shows an example decision process, which maps the output of a classification ML API on an image to the image's recycling categorization specific to this application.

Critical and non-critical errors: For a given decision process, some API output errors will still lead to a correct decision, whereas some API output errors will lead to an incorrect decision and hence an application failure. We refer to the former as *non-critical errors*, and the latter as *critical errors*.

2.2 Methodology

Our work focuses on applications that use ML API output to make control-flow decisions. To this end, we look at 77 open-source applications which collectively use six widely used vision and language APIs [10,65] offered by two popular cloud AI service providers, as summarized in Table 1.

These applications come from two sources. First, we study all 50 applications that use vision and language APIs from a recently published benchmark suite of open-source ML applications [66]. Second, given the popularity of image classification APIs [11, 12], we additionally sample 27 applications from GitHub that use Google and Amazon image classification APIs (16 for the former and 11 for the latter). We obtain these 27 by checking close to 100 applications that use image classification APIs and filtering out those that directly print out or store the API output. Every application in our benchmark suite uses exactly one ML API for decision making.

Threats to validity: While many applications use the APIs listed in Table 1, there are a few other APIs not covered in our study. A few vision and language-related ML tasks are not as popular and hence are not covered in our study (*e.g.,* face recognition and syntax analysis). Speech APIs are not covered, because their outputs are rarely used to affect application control flow based on our checking of open-source applications. Finally, our study does not cover applications that use ML APIs offered by other cloud or local providers.

2.3 Understanding the decision mechanism

Q1*: What types of ML API outputs are typically used by applications to make decisions?*

ML APIs produce output of a variety of types. The sentiment analysis API outputs a list of floating-point value pairs (`score` and `magnitude`), describing the sentiment of the whole document and every individual sentence; the other five APIs in Table 1 each produces a list of categorical labels ranked in descending order of their confidence scores, which is also part of the output. Some APIs' output also contains other information, like coordinates of bounding boxes, entity names, links to Wikipedia URLs, and so on. Among all these, only two types have been used in application decision processes of our studied application: the floating-point pair (`score` and `magnitude`) and the categorical labels.

For the 63 applications that use categorical-label output from the five APIs (all except `analyze_sentiment` in Table 1), they each define one or more *label lists* and check which label list(s) an API output label belongs to. The code snippet of a landmark classification application in Figure 2(a) is an example of this. It calls the `label_detection` API with a sight-seeing image and checks the output labels to see if the image might contain `Landmark`, or just ordinary `Building`, or `Person`.

For the 14 applications that use the `analyze_sentiment` API, they each define several *value ranges* and check which range the sentiment `score` and/or `magnitude` falls in. The code snippet of `FoodDelivery` [48] in Figure 2(b) is an example. This application calls `analyze_sentiment` with a restaurant review text, and then checks the returned sentiment `score` to judge if the review is negative, positive, or neutral.

Q2*: What type of decisions do applications make?*

We observe three categories of ML-based decision making, which we name following common question types in exams:

(1) True-False decision, where a single label list or value range is defined and one selection is allowed: either the ML API output belongs to this list/range or not. This type occurs

Invocation of ML API | Branch condition that uses API output | Structure indicating decision types

```
Landmark = ["Landmark", "Sculpture"]
Building = ["Building", "Estate",
            "Mansion"]
Person = ["Person", "Lady"]
res = client.label_detection(Img)
annotations = res.label_annotations
labels = [obj.name for obj in annotations]
if any([l in Landmark for l in labels]):
  return "Landmark"
elif any([l in Building for l in labels]):
  return "Building"
elif any([l in Person for l in labels]):
  return "Person"
else: ## obj.name not in any list
  return "It is others."
```

(a) `Aander-ETL`
(Label output, Multi-Choice, App-Order)

```
text = types.Document(content=Text)
res = client.analyze_sentiment(text)
sentiment = res.document_sentiment
sentiment_score = sentiment.score
if sentiment_score < 0.3:
  print("It's a negative sentence!")
elif sentiment_score > 0.6:
  print("It's a positive sentence!")
else: ## between 0.3 and 0.6
  print("It's a neutral sentence!")
```

(b) `FoodDelivery`
(Float-point value output, Multi-Choice)

```
Plant = ["Houseplant","Bonsai",
         "Plant", "Flowerpot"]
res =
   client.label_detection(Img)
for obj in
response.label_annotations:
  if obj.name in Plant:
    return "Plant found!"
return "No plant found."
```

(c) `Plant-watcher`
(Label output, True-False)

```
Protein = ["Hamburger","Meat",
           "Patty"]
Grain = ["Noodle","Pasta","Bread"]
Fruit = ["Apple","Orange","Pear"]
res = client.label_detection(Img)
returned_set = set()
for obj in res.label_annotations:
  if obj.name in Protein:
    returned_set.add("protein")
  elif obj.name in Grain:
    returned_set.add("grain")
  elif obj.name in Fruit:
    returned_set.add("fruit")
return returned_set
```

(d) `The-Coding-Kid`
(Label output, Multi-Select)

Figure 2: *Code snippets from five example applications where ML API output affects control flow decisions in different ways.*

in about one third of the applications in our study. For example, the plant management application `Plant-watcher` [57] (Figure 2(c)) checks to see if the image contains plants or not.

(2) Multi-Choice decision, where multiple lists of labels or value ranges are defined, and one selection is allowed. The ML API output will be assigned to *at most one* list or range; the application's decision making logic determines which of these lists/ranges the output belongs to, or determines that the output belongs to none of them. This type of decision is the most common, occurring in about 45% of benchmark applications. The garbage classification application discussed in §1 makes such a Multi-Choice decision. It decides which one of the following classes the input image belongs to: `Recycle`, `Compost`, `Donate`, or none of them.

(3) Multi-Select decision, where multiple label lists or value ranges are defined, and *multiple* selections are allowed about which label lists or value ranges the ML API output belongs to. This type of decisions occur in close to a quarter of the applications. Figure 2(d) illustrates such an example from the nutrition advisor application `The-Coding-Kid` [62]. This application defines three label lists to represent nutrition types: `Protein`, `Grain`, and `Fruit`, and it checks to find all the nutrition types present in the input image.

In the remainder of the paper, we will use **target class** to refer to a label list (or a value range) that is used to match against a categorical label (or a value). For instance, the code snippet in Figure 2(a) has three label lists as its target classes (`[Landmark, Sculpture]`, `[Building, Estate, Mansion]`, and `[Person, Lady]`), and the code snippet in Figure 2(b) has three value ranges as its target classes (`<0.3`, `>0.6`, and in between).

***Q3**: How do applications reach Multi-Choice decisions?*

When the ML API outputs multiple labels, the outcome of a Multi-Choice decision varies depending on which *matching order* is used. First, the matching order can be determined by the API output. For example, the garbage classification application (Figure 1) first checks whether the first label in the API output matches any target class. If so, later API output labels will be skipped, even if they might match with a different class. If there is no match for the first label, the second output label is checked, and so on. These labels are ranked by the API in the descending order of their associated confidence scores, so we refer to such a matching order as API-order. It is used by 80% of applications that make Multi-Choice decisions.

The matching order can also be specified by the application, referred to as App-order. For instance, regardless the API output, application `Aander-ETL` [1] (Figure 2(a)) always first checks if the `Landmark` class matches with *any* output label. If there is a match, the decision is made. Only when it fails to match `Landmark`, will it move on to check the next choice, `Building`, and so on. This matching order is used by 20% of applications that make Multi-Choice decisions.

2.4 Understanding the decision implication

***Q4**: Does an application need ML APIs that can accurately identify thousands of labels?*

ML models behind popular ML APIs are well trained to support a wide range of applications. For example, Google and Microsoft's image-classification APIs are capable of identifying more than 10000 labels [44], while Amazon's image-classification API can identify 2580 labels [3]. However, for each individual application, its decision making only requires classifying the input image into a handful of target classes: 7 at most in our benchmark applications. The largest number of image-classification labels checked by an application is 54, a tiny portion of all the labels an image-classification API could output.

Clearly, for any application, a customized ML model that focuses on those target classes used by the application's decision process has the potentially to out-perform the big and generic ML model behind ML APIs. How to accomplish the customization without damaging the accessibility of ML APIs will be the goal of ChameleonAPI.

***Q5**: Are there equivalence classes among ML API outputs in the context of application decision making?*

For the 63 applications that make decisions based on API output of categorical labels, they present 121 target classes in total, each containing 4.7 labels on average (3 being the median). Only 35 target classes in 22 applications contain a single label. For the 14 applications that make decisions based on

floating-point sentiment `score` and `magnitude`, their target classes *all* contain an infinite number of `score` or `magnitude` values. In other word, no class contains just a single value.

Clearly, the wide presence of multi-value target classes creates equivalence classes among output returned by the API—errors within one equivalence class are *not* critical to the corresponding application. This offers another opportunity for ML customization.

***Q6**: How much difference is there between different applications' target classes?*

Overall, the difference is significant. We have conducted pair-wise comparison between any two applications in our benchmark suit, and found that 88% of application pairs share *no* common labels in any of their target classes. Similarly, among the 381 labels that appear in at least one application's target classes, 88% of them appear in only one application (*i.e.,* 335 out of 381 labels).

Clearly, there is *little overlap* among the target classes of different applications, again making a case for *per-application* customization of the ML models used by the ML APIs.

***Q7**: Do different decision mechanisms imply different sensitivity to output errors of ML APIs?*

Even for two applications that have the same target classes, if they try to make different types of decisions, they will have different sensitivity to ML API output errors—some API errors might be critical to one application, but not to the other. For example, errors that affect the selection of different target classes are equally critical to Multi-Select decisions. However, this is not true for Multi-Choice, where only the first matched target class matters. Furthermore, the matching order of a Multi-Choice decision affects which errors are critical. When the API-output order is used (*e.g.,*`HeapsortCypher` in Figure 1), an error on the first label in the API output is more likely to be critical than an error on other labels in the output. However, when App-order order is used (*e.g.,*`Aander-ETL` in Figure 2(a)), errors related to labels in the first target class (*e.g.,*`Landmark`) are more likely to be critical than those related to labels in later target classes (*e.g.,*`Person`).

Clearly, to customize ML models for each application, we need to take into account what is the decision type and what is the matching order (for Multi-Choice decisions).

3 Design of ChameleonAPI

Inspired by the study of §2, we now present ChameleonAPI which automatically customizes ML models for applications.

3.1 Problem formulation

Goal: For an application that uses ML APIs, our goal is to **minimize critical errors** in the API outputs for this application by efficiently re-training the original generic neural network models underneath these APIs into customized models; our approach stands in contrast to typical approaches that minimize *all* inference errors. In other words, the new ML model should return outputs that lead the application process to the same decision as if the ground-truth of the input is returned by the ML API.

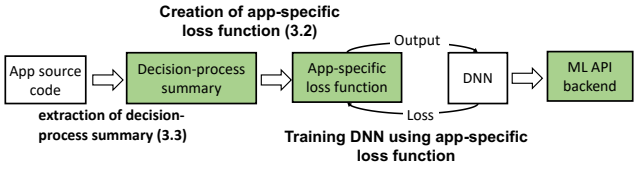


Figure 3: *The logical steps of how ChameleonAPI customizes d for individual applications.*

To formally state this objective, we denote how an application makes a decision by $App(API(\mathbf{x}))$, where $\mathbf{x}$ is the input to the ML API and $API(\mathbf{x})$ is the API output. Then for a given application decision process of $App(\cdot)$ and an input set $\mathbf{X}$[1], our goal is to train an ML model $DNN(\cdot)$ such that

$$\min_{\mathbf{x}_i \in \mathbf{X}} \left| \{\mathbf{x}_i | App(API(\mathbf{x}_i)) \neq App(\widehat{API}(\mathbf{x}_i))\} \right|, \quad \text{where } API(\mathbf{x}_i) = F(DNN(\mathbf{x}_i)) \tag{1}$$

Here, $\widehat{API}(\mathbf{x}_i)$ is a hypothetical API function that always returns the ground truth of input $\mathbf{x}_i$, and $F(\cdot)$ represents the postprocessing used by the API to translate a DNN output to an API output. For instance, an image classification model's output is a vector of confidence scores between 0 and 1 (each for a label), but the ML API will use a threshold θ to filter and return only labels with scores higher than θ, or the top k labels with the highest confidence scores.

Our goal in Eq 1 differs from the traditional goal of an ML model, which minimizes any errors in the API output, *i.e.,*

$$\min_{\mathbf{x}_i \in \mathbf{X}} \left| \{\mathbf{x}_i | API(\mathbf{x}_i) \neq \widehat{API}(\mathbf{x}_i)\} \right|. \tag{2}$$

Given that it is hard to obtain a DNN with 100% accuracy, the difference between the two formulations is crucial, since not all API output errors in Eq. 2 will cause incorrect application decisions in Eq. 1. Thus, compared to optimizing Eq. 2, optimizing Eq. 1 is more likely to focus the DNN training on reducing the critical errors for the application.

To train a DNN that optimizes Eq. 1, we need to decide if a DNN inference output $DNN(\mathbf{x})$ is a critical error or not (*i.e.,* $App(DNN(\mathbf{x})) \neq App(\widehat{API}(\mathbf{x}))$) at the end of *every* training iteration. This decision needs to be made automatically and efficiently. For example, repeatedly running the entire ML application after every training iteration would not work, as it may significantly slow down the training procedure.

[1] A careful reader might notice that the formulation in Eq. 1 also depends on the input set. Though the input set should ideally follow the same distribution of real user inputs of the application, this distribution is hard to obtain in advance and may also vary over time and across users. Instead, we focus our discussion on training the ML model to minimize Eq. 1 with an assumed input distribution. Our evaluation (§5) will test the resulting model's performance over different input distributions.

Logical steps of ChameleonAPI: To customize and deploy the DNN for an application, ChameleonAPI takes three logical steps (Figure 3). First, ChameleonAPI extracts from an application's source code a *decision-process summary* (explained shortly), a succinct representation of the application's decision process, which will be used to determine if a DNN inference error is critical (details in §3.3). Second, ChameleonAPI converts a decision-process summary to a *loss function*, which can be directly used to train a DNN (details in §3.2). This loss function only penalizes DNN outputs that lead to critical errors with respect to a given application. Finally, the loss function will be used to train a customized DNN for this particular application's ML API invocations (§3.4).

A decision-process summary is a succinct abstraction of the application that contains enough information to determine if a DNN inference output causes a critical error or not. Specifically, it includes three pieces of information (defined in §2.3):

- *Composition of target classes:* the label list or value range of each target class;
- *Decision type:* True-False, Multi-Choice, or Multi-Select;
- *Matching order:* over the target classes, API-order or App-order, if the application makes a Multi-Choice decision.

For a concrete example, the decision-process summary of the garbage classification application in Figure 2(a) contains (1) three label lists representing three target classes: `Recycle`, `Compost`, and `Donate`; (2) the Multi-Choice type of decision; and (3) the matching order of API-order.

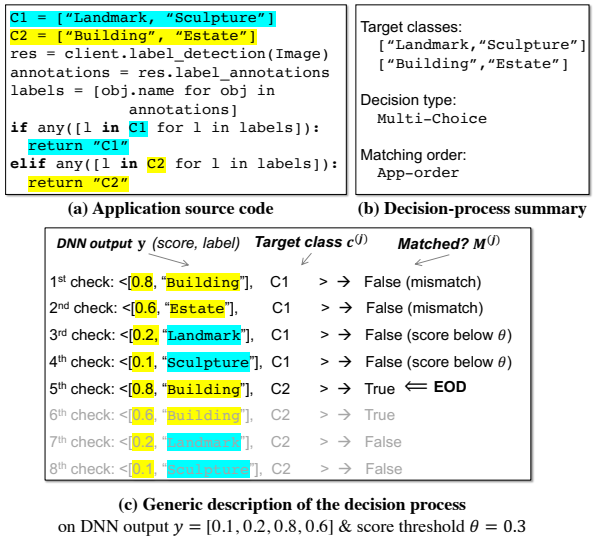
What is changed, what is not: ChameleonAPI does *not* change the ML API or the application source code. Unlike recent work that aims to shrink the size of DNNs or speed them up [36, 37, 54], we do not change the DNN architecture (shape and input/output interface); instead, we train the DNN to minimize critical errors. That said, deploying ChameleonAPI has two requirements. First, the application developers need to run ChameleonAPI's parser script to automatically extract the decision-process summary. Second, an ML model needs to be retrained for each application, instead of serving the same model to all applications.

The remainder of this section will begin with the design of the application-specific loss function based on decision-process summary, followed by how to extract the decision-process summary from the application, and finally, how the customized ML models are used to serve ML API queries.

3.2 Application-specific loss function

Given Eq 1, ChameleonAPI trains a DNN model with a *new loss function*, which only penalizes critical errors of an application, rather than all DNN inference errors. Since decision processes vary greatly across applications (§2.4), we first explain how to conceptually capture different decision processes in a generic description, which allows us to derive the mathematical form of ChameleonAPI's loss function later.

Generalization of decision processes: For each application



(a) Application source code (b) Decision-process summary

(c) Generic description of the decision process on DNN output $y = [0.1, 0.2, 0.8, 0.6]$ & score threshold $\theta = 0.3$

Figure 4: *The generic description (shown in (c)) of an application (whose source code is shown in (a) and decision-process summary in (b)) on a DNN inference output* **y**.

in our study (§2.2), our insight is that its decision process can always be viewed as traversing a sequence of conditional *checks* until an *end-of-decision* (*EOD*) occurs:

$$
\begin{aligned}
&1^{\text{st}} \text{ check: } < \mathbf{y}, c^{(1)} > \rightarrow M^{(1)} \\
&\quad \ldots \\
&j^{\text{th}} \text{ check: } < \mathbf{y}, c^{(j)} > \rightarrow M^{(j)} \qquad \Leftarrow \textbf{EOD} \\
&\quad \ldots
\end{aligned}
$$

where the j-th check takes as input the DNN output **y** and one of target classes $c^{(j)}$, and returns a binary $M^{(j)}$ indicating whether $\mathbf{y}^{(j)}$ *matches* the condition of $c^{(j)}$ and a binary decision whether this check happens before the EOD. The set of target classes successfully matched before the EOD will be those selected by the application.

Figure 4 shows (a) an example application, (b) the decision-process summary, and (c) the generic description for this application's decision process and a DNN output.

This generic description (*e.g.,* the traversal order of the target classes, how a match is determined in a check, and when the EOD occurs) will depend on the information in the decision-process summary and the DNN output **y**. We stress that this generic description may *not* apply to all applications, but it does apply to all applications in our study (§2.2).

Categorization of critical errors: Importantly, this generic description helps to categorize critical errors:

- *Type-1 Critical Errors:* A correct target class c is not matched before EOD, but will be so if EOD occurs later.
- *Type-2 Critical Errors:* A correct target class c is never matched, before or after the EOD.
- *Type-3 Critical Errors:* An incorrect target class c is matched before EOD.

A useful property of this categorization is that any wrong decision (a correct target class not being picked, or an incorrect target class being picked) falls in a unique category, and non-critical errors do not belong to any category. In other words, as long as the loss function penalizes the occurrences of each category, it will only capture critical errors.

ChameleonAPI's first attempt of a new loss function: To understand why it is difficult to penalize critical errors and critical errors *only*, we first consider the common practice of assigning a higher weight to the loss of a DNN output if the ground-truth of the input will lead to a selection of some target classes (*e.g.,* [26, 35, 64]). Henceforth, we refer to this basic design of loss function as ChameleonAPI$_{basic}$.

At best, ChameleonAPI$_{basic}$ might improve the DNN's *label-wise* accuracy on inputs whose ground-truth decision selects some target classes. However, as elaborated in §2.3, we also need to consider which labels belong to the same target class, the decision type, and the matching order of an application decision process in order to capture the three types of critical errors. For instance, in the garbage-classification application (Figure 1), without knowing the label lists of each target class, ChameleonAPI$_{basic}$ will give an equal penalty to a critical error of mis-classifying a `Paper` image to `Wood` and a non-critical error of mis-classifying a `Paper` image to `Shirt`. Similarly, without knowing the matching order, ChameleonAPI$_{basic}$ will equally penalize the output of [`Plastic`, `Jacket`] and [`Jacket`, `Plastic`], but only the latter leads to correct output because `Jacket` is matched first.

ChameleonAPI's loss function: ChameleonAPI leverages the categorization of critical errors to systematically derive a loss function that penalizes each type of critical error. To make it concrete, we explain ChameleonAPI's loss function of "label-based API, Multi-Choice type of decision, and App-order" (*e.g.,* Figure 4). Appendix§B will detail the loss functions of other decision processes. The loss function of such applications has three terms, each penalizing one type of critical error:

$$L(\mathbf{y}) = \overbrace{\text{Sigmoid}\left(\min\left(\max_{l\in\cup_{c<\hat{c}}G_c}\mathbf{y}[l], \max_{l\in G_{\hat{c}}}\mathbf{y}[l]\right)-\theta\right)}^{\textbf{Type-1 Critical Errors}} + \overbrace{\text{Sigmoid}\left(\theta-\max_{l\in G_{\hat{c}}}\mathbf{y}[l]\right)}^{\textbf{Type-2 Critical Errors}} + \overbrace{\sum_{c<\hat{c}}\text{Sigmoid}\left(\max_{l\in G_c}\mathbf{y}[l]-\theta\right)}^{\textbf{Type-3 Critical Errors}} \quad (3)$$

Here, $\mathbf{y}[l]$ denotes the score of the label l, G_c denotes the set of labels of target class c, $\hat{c}$ denotes the correct (*i.e.,* ground-truth) target class, and the sigmoid function $\text{Sigmoid}(x) = \frac{1}{1+e^x}$ will incur a higher penalty on a greater positive value.

Why does it capture the critical errors? Given this application is Multi-Choice, the EOD will occur right after the first match of a target class, *i.e.,* the first check with a c such that $\max_{l\in G_c}\mathbf{y}[l] \geq \theta$.

- A Type-1 critical error occurs, if (1) the correct target class $\hat{c}$ is matched *and* (2) it is matched after the EOD. First, the correct target class $\hat{c}$ is matched, if and only if at least one of its labels has a score above the confidence threshold, so $\max_{l\in G_{\hat{c}}}\mathbf{y}[l] \geq \theta$). Second, this match happens after the break, if and only if some target class c before $\hat{c}$ (*i.e.,* $c < \hat{c}$) is matched, so $\max_{l\in G_c}\mathbf{y}[l] \geq \theta$). Put together, the first term of Eq 3 penalizes any occurrence of these conditions.
- A Type-2 critical error occurs, if no label in the correct target class $\hat{c}$ has a score high enough for $\hat{c}$ to be matched, *i.e.,* $\max_{l\in G_{\hat{c}}}\mathbf{y}[l] < \theta$, so the second term of Eq 3 penalizes any occurrence of this condition.
- A Type-3 critical error occurs, if any incorrect target class c before $\hat{c}$ (*i.e.,* $c < \hat{c}$) has a label with a score high enough for c to be matched, *i.e.,* $\max_{l\in G_c}\mathbf{y}[l] - \theta$, so the third term of Eq 3 penalizes any occurrence of this condition.

To train a DNN, the loss function must be differentiable with respect to the DNN ouput $\mathbf{y}$. Eq 3 uses the max function several times. Though max is not naturally differentiable, it can be closely approximated in well-known differentiable forms provided by PyTorch's differentiable operators [56]).

3.3 Extracting applications' decision process

The current prototype of ChameleonAPI program analysis supports Python applications that make decisions based on categorical label output or floating point output of ML APIs. We first discuss how it works for ML APIs with categorical label output, like all the APIs in Table 1 except for `analyze_sentiment`. We will then discuss a variant of it that works for most use cases of `analyze_sentiment`.

Given application source code, ChameleonAPI first identifies all the invocations of ML APIs. For every invocation I in a function f, ChameleonAPI then identifies all the branches whose conditions have a data dependency upon the ML API's label output. We will refer to these branches as I-branches. If there is no such branch in f, ChameleonAPI then checks the call graph, and analyzes up to 2 levels of callers and up to 5 levels of callees of f until such a branch is identified. If no such branch is identified after this, ChameleonAPI considers the ML API invocation I to not affect application decisions and hence does not consider any optimization for it. If some I-branches are identified, ChameleonAPI records the top-level function analyzed, F, and moves on to extract the decision-process summary in following steps.

What are the target classes? ChameleonAPI figures out all the target classes and their composition in two steps.

The first step leverages symbolic execution and constraint solving to identify all the labels that belong to *any* target classes. Specifically, ChameleonAPI applies symbolic execution to function F, treating the parameters of F and the label output of I as symbolic (*i.e.,* the symbolic execution skips the ML API invocation I and directly uses I's symbolic

output in the remaining execution of F)[2]. Since applications typically match only one label in API output at a time (as observed in §2.3), we set the label array returned by I to contain one element (label) and use a symbolic string to represent it. Through symbolic execution, ChameleonAPI obtains constraints for every path that involves an I-branch, solving which tells ChameleonAPI which labels need to be in the output of the ML API in order to execute each unique path, essentially all the labels that belong to any target class.

One potential concern is that a solver may only output one instead of all values that satisfy a constraint. Fortunately, the symbolic execution engine used by ChameleonAPI, NICE [31], turns Python code into an intermediate representation where each branch is in a simplest form. Take Figure 2(d) as an example, the source-code branch `if obj.name in Protein` is transformed into three branches where `obj.name` is compared with `"Hamburger"`, `"Meat"`, and `"Patty"` separately, allowing us to capture all three labels by solving three separate path constraints.

The second step groups these labels into target classes by comparing their respective paths: if two API output, each with one label, lead the program to follow the same execution path at the source-code level, these two labels belong to the same target class. For example, in Figure 2(d), the execution path is exactly the same when the `label_detection` API returns `["Hamburger"]`, comparing with when it returns `["Meat"]`, with all function parameters and other API output fields being the same. Consequently, we can know that label `Hamburger` and label `Meat` belong to the same target class. To figure out the path, ChameleonAPI simply executes function F using each input produced by the constraint solver and traces the source-code execution path using the Python trace module.

One final challenge is that ChameleonAPI needs to identify and exclude the path where none of the target classes are matched (*e.g.,* the `"It is others."` path in Figure 2(a)). We achieve this by carefully setting the default solution in the constraint solver to be an empty string, which is impossible to output for any ML APIs in this paper. This way, whenever this default solution is output, ChameleonAPI knows that the corresponding path matches no target class.

What is the type of decision? When only one target class is identified, ChameleonAPI reports a True-False decision type. Otherwise, ChameleonAPI decides whether the decision type is Multi-Choice or Multi-Select by checking the source-code execution path associated with every target class label obtained above. If any execution evaluates an I-branch *after* another I-branch is already evaluated to be true, ChameleonAPI reports a Multi-Select decision type; otherwise, ChameleonAPI reports a Multi-Choice decision type.

What is the matching order over the target classes? To tell whether a Multi-Choice decision is made through API-Order like in Figure 1 or App-Order like in Figure 2(a), ChameleonAPI first identifies all the `for` loops that iterate through the label array output by the ML API and have control-dependency with I-branches, *e.g.,* the `for l in labels` in Figure 2(a) and the `for obj in response.label_annotations` in Figure 1.

ChameleonAPI then checks how many such output-iterating loops there are. If there is only one and this loop is not inside another loop, like that in Figure 1, ChameleonAPI considers the matching order to be API-Order, as the application only iterates through each output label once, with the matching order determined by the output array arranged by the ML API. Otherwise, ChameleonAPI considers the matching order to be App-Order. This is the case for the example shown in Figure 2(a), where three output-iterating loops are identified, each of which matches with one target class in an order determined by the application: the `Landmark` target class, followed by the `Building`, and finally the `Person`.

How to handle floating-point output of ML APIs? Recall in §2.3 that some ML APIs, *e.g.,*`analyze_sentiment`, have floating-point output and the application defines several value ranges to put each floating-point output into one category. To handle this type of API, ChameleonAPI needs to identify the value range of each target class, which is not supported by NICE and other popular constraint solvers. Fortunately, many applications directly compare API output with constant values in I-branches, giving ChameleonAPI a chance to infer the value range. For these applications, ChameleonAPI first extracts those constant values that are compared with API output in I-branches, *e.g.,* 0.3 and 0.6 in Figure 2(b). ChameleonAPI then forms tentative value ranges using these numbers, like -1 – 0.3, 0.3 – 0.6, and 0.6 – 1 for Figure 2(b) (-1 and 1 are the smallest and biggest possible `score` output of `analyze_sentiment` based on the API manual). To confirm these value ranges and figure out the boundary situation, ChameleonAPI then executes function F with all the boundary values, as well as some values in the middle of each range. By comparing which values lead to the same execution path, ChameleonAPI finalizes the value ranges. For the example in Figure 2(b), after executing with `score` set to -0.35, 0.3, 0.45, 0.6, and 0.8, ChameleonAPI settles down on the final value ranges to be: (-1,0.3), [0.3,0.6), and [0.6,1).

Limitation The static analysis in ChameleonAPI does not handle the iterated object of while loops, unfolded loops, and recursive functions. For complexity concerns, ChameleonAPI only checks caller and callee functions with limited levels, and hence may miss some I-branches far away from the API invocation. ChameleonAPI's ability of identifying target classes is limited by the constraint solver. ChameleonAPI assumes different source-code paths correspond to different target classes, which in theory could be wrong if the application behaves exactly the same under different execution paths.

[2]Recall that an API output contains several fields not used to influence control flow in any applications. We set them with pre-defined dummy values.

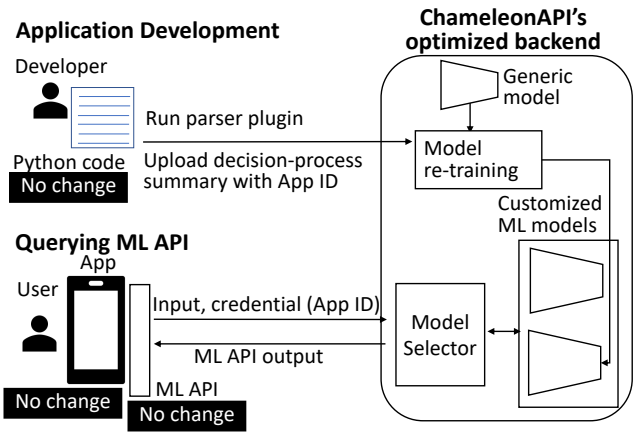


Figure 5: *Workflow of ChameleonAPI.*

3.4 Putting them together

We put these components together into a ML-as-a-Service workflow shown in Figure 5.

First, when an application (A) is developed or updated, the developers run a parser (described in §3.3) provided by ChameleonAPI on A's source code to extract the decision-process summary for A. The developers can then upload the decision-process summary to ChameleonAPI's backend together with a unique application ID[3] (which will later be used to identify queries from the same application).

ChameleonAPI's backend then uses the received decision-process summary to construct a new application-specific loss function (described in §3.2). When a DNN is trained using the new loss function, its inference results will lead to fewer critical errors (*i.e.,* incorrect application decisions) for application A. In our prototype, ChameleonAPI uses the new loss function to re-train an off-the-shelf pre-trained DNN, a common practice to save training time (see §5 for quantification). The DNN re-training uses an application-specific dataset sampled from the dataset used by the pre-trained generic DNN (see Table 2 and §4), so that each target/non-target class is selected by ground-truth decisions of the same number of inputs.

Finally, ChameleonAPI backend maintains a set of DNN models, each customized for an application and keyed by the application ID. When application A invokes an ML API at run time, the ChameleonAPI backend will use the application ID associated with the API query to identify the DNN model customized for A, run the DNN on the input, and return the inference result of the selected model to the application.

Note that, ChameleonAPI can also be used to customize ML models that run locally behind the ML APIs, instead of those in the cloud through ML service providers. In this case, developers run the ChameleonAPI parser on their application and save the parser's result into a local file. This local file will then be consumed to help re-train an off-the-shelf DNN into a customized DNN to serve the application.

[3] In many MLaaS offerings [2, 20], a connection between the application and the MLaaS backend is commonly created before the application issues any queries. Existing MLaaS already allows applications to specify the application ID via the connection between the application and backend.

	Dataset	Generic model
Image Classification	OpenImages [44]	TResNet-L [6]
Object Detection	COCO [14]	Faster-RCNN [58]
Sentiment Analysis	Amazon review [39]	BERT [18]
Text Classification	Yahoo [30]	BERT [18]
Entity Recognition	conll2003 [63]	BERT [18]

Table 2: *The ML APIs and datasets in evaluation.*

4 Implementation

Extractor of decision-process summary: The current prototype of ChameleonAPI is implemented for Python applications that use Google or Amazon ML APIs. It takes as input the application source code and returns as output the decision-process summary in the JSON format. It uses NICE symbolic execution engine [31] and CVC5 constraint solver [5] to identify target classes, and uses Python static analysis framework Pyan [47] and Jedi [24] to identify the decision type and the matching order. Particularly, it identifies the object that is iterated through by a `for`-loop through the `iter` expression in each for-loop header, which is used to distinguish Multi-Choice and Multi-Select decisions and the matching order.

ML re-training: The re-training module is implemented in PyTorch v1.10 and CUDA 11.1. It uses a decision-process summary to construct a new loss function (see §3.2), and then replaces the builtin loss function in Pytorch with the new loss function, and uses the common forward and backward propagation procedure to re-train an off-the-shelf pre-trained DNN model (explained next).

Generic models: Without access to the models and the training data used by commercial ML services, we use open-sourced pre-trained DNNs and their training datasets as a proxy, which are summarized in Table 2. These DNNs are trained on the "training" portion of their respective datasets. They are trained to achieve good accuracy over a wide range of labels, and we have confirmed that their accuracies in terms of application decisions are similar to the real ML APIs (§5.2).

Training data: We make sure that the labels included in these datasets cover the labels used in the decision processes of the applications in our study. An exception is text classification: to our best knowledge, there is no open-source dataset that covers the classes in Google's text classification API. Instead, we use the Yahoo Question topic classification dataset [30], whose classes are similar to those used in the applications.

Instead of training DNNs on all training data, most of which do not match any target classes of an application, we create a downsampled training set for ChameleonAPI and ChameleonAPI$_{basic}$. For each application, we randomly sample (without replacement) its training data such that each target class and the non-target class (not matching any target

class) is the correct decision for the same number of training inputs, which depending on applications, ranges from 12K to 40K. With such training set, ChameleonAPI$_{basic}$ will be equivalently implemented by training on the downsampled training set using the conventional loss function (*i.e.,* cross-entropy loss for classification tasks). Moreover, the downsampled training set significantly speedups DNN re-training (§5.2).

5 Evaluation

Our evaluation aims to answer following questions: How much can ChameleonAPI reduce incorrect application decisions? How long does it take ChameleonAPI to customize DNN models for applications? and Why does ChameleonAPI reduce incorrect application decisions where ChameleonAPI$_{basic}$ falls short?

5.1 Setup

Applications: We have applied ChameleonAPI on all the 77 applications summarized in Table 1. Due to space constraints, our discussion below focuses on the 57 applications that involve three popular ML tasks, image-classification, object-detection, and text-classification, and omits the remaining 20 applications that involve sentiment analysis and entity recognition. The results of the latter show similar trends of advantage from ChameleonAPI and are available in Appendix §D.

Metrics: For each scheme (explained shortly) and each application, we calculate the *incorrect decision rate* (*IDR*): the fraction of testing inputs whose application decisions do not match the correct application decisions (*i.e.,* decisions based on human-annotated ground truth).

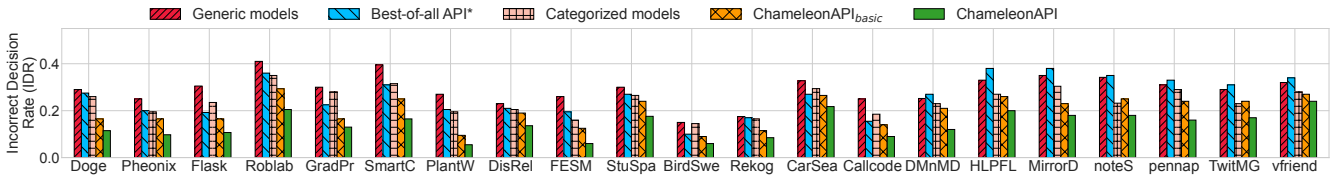
Schemes: We compare the results of these schemes:

- *Various commercial ML APIs*: the results returned by ML APIs of three service providers (Google [20], Amazon [2] and Microsoft [51]).
- *Best-of-all API**: a hypothetical method that queries ML APIs from those three service providers on each input and picks the best output based on the classic definitions of accuracy: label-wise recall for classification tasks and mean-square-error of floating-point output for sentiment analysis. This serves as an idealized reference of recent work [11,12], which tries to select the best API output with high label-wise accuracy.
- *Generic models*: the open-sourced generic model based on which the next three schemes are re-trained. They serve as a reference without customization and achieve similar accuracy as commercial APIs. Their details are explained in Section 4.
- *Categorized models*: This scheme pre-trains a number of specialized models. Each specialized model replaces the last layer of the generic model so that it outputs the confidence scores for a smaller number of labels representing a common category (e.g., "dog", "animal", "person" and a few other labels represent the "natural object" category), and is fine tuned from the generic model accordingly. A simple parser checks which labels are used by an application. If all the labels belong to one category, the corresponding model specialized for this category is used to serve API calls from this application. If the labels belong to multiple categories, multiple specialized models will be used, which we will explain more later. We set up 35 categories for image classification and 7 categories for object detection based on the Wikidata knowledge graph [68], as well as 15 categories for text classification based on the inherent hierarchy in Google text-classification output. More details of how we have designed these categories are available in the Appendix §C.

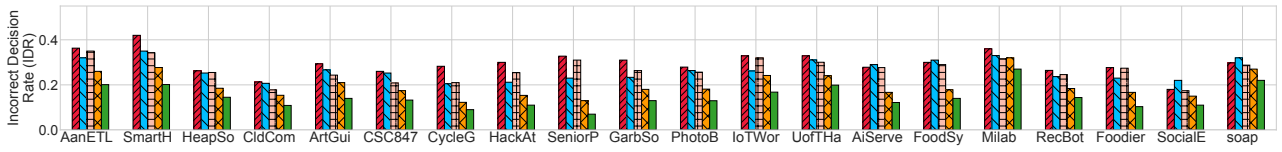
  Note that, we have designed this scheme to represent a middle-point in the design space between the generic model and the ChameleonAPI approach: on one hand, this scheme offers some application customization, but not as much as ChameleonAPI (e.g., which labels belong to the same target class, what is the decision process, and what is the matching order used by the application are all ignored); on the other hand, this scheme requires a simpler parser compared to ChameleonAPI.
- *ChameleonAPI$_{basic}$*: the model is re-trained with ChameleonAPI's training data, which concentrates on labels used by the application, but with the conventional loss function. Like Categorized models, this scheme only needs a simple parser that extracts which labels are used by the application, and does not make use of other application information that ChameleonAPI uses. Unlike Categorized models, this scheme prepares a customized model for each application, instead of relying on a small number of categorized models.
- *ChameleonAPI* (our solution): the model re-trained with our training data and loss function.

Testing data: For the same application, all schemes are tested against the same testing input set. The testing set of an application is randomly sampled from the "testing" portion of the dataset associated with the application's generic model (Table 2). We make sure that *no* testing input appears in the training data. Like the creation of training data of ChameleonAPI (§4), by default, we randomly sample the testing data such that each target class and the non-target class (not matching any target class) appear as the correct decision for the same number of testing inputs, which ranges from 1.2K to 4K. This is similar to the testing sets used in related work on ML API (*e.g.,* [11,36,37,66]). Such data downsampling is commonly used in ML [19,46]. Other than Figure 9, we will use this as the default testing dataset.

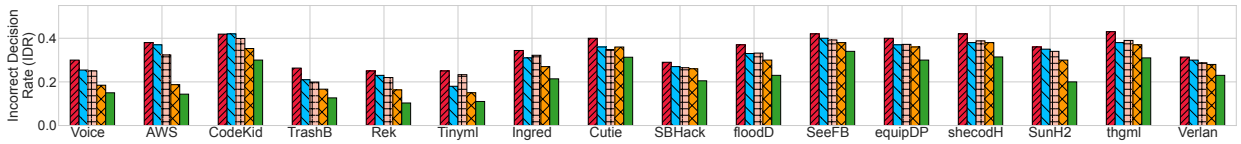
Hardware setting: We evaluate ChameleonAPI and other approaches on a GeForce RTX 3080 GPU, and an Intel(R) Xeon(R) E5-2667 v4 CPU, with 62GB memory.



(a) Applications making True-False decisions.



(b) Applications making Multi-Choice decisions.



(c) Applications making Multi-Select decisions.

Figure 6: *ChameleonAPI reduces the incorrect decision rate (IDR) on the 57 applications that use Google's or Amazon's image-classification, text-classification, and object-detection APIs.*

	True-False	Multi-Choice	Multi-Select
Google API	0.29	0.32	0.35
Microsoft API	0.30	0.33	0.32
Amazon API	0.31	0.33	0.36
Best-of-all API*	0.26	0.27	0.31
Generic models	0.29	0.30	0.34
Categorized models	0.24	0.27	0.31
ChameleonAPI$_{basic}$	0.19	0.22	0.27
ChameleonAPI	**0.13**	**0.16**	**0.21**

Table 3: *Average incorrect decision rate (IDR) among apps that make different types of decisions. The lower the better. The top half represents commercial APIs and their idealistic combinations; the bottom half represents open-source models.*

	Single-category	Multi-category
Generic models	0.32	0.28
Categorized models	0.28	0.27
ChameleonAPI$_{basic}$	0.24	0.18
ChameleonAPI	0.17	0.14

Table 4: *Average IDR among single-category and multi-category applications. The lower the better.*

5.2 Results

Overall gains: Measured by the average incorrect decision rate (IDR) across all applications, the most accurate scheme is ChameleonAPI, with an IDR of 0.16, and the least accurate scheme is Generic models, with an IDR of 0.31. In other words, ChameleonAPI successfully reduces the number of incorrect decisions of its baseline model by almost 50%. ChameleonAPI$_{basic}$ (0.22), Categorized models (0.28), and Best-of-all API* (0.28) have IDR rates in between.

The advantage of ChameleonAPI, and even ChameleonAPI$_{basic}$, over the other schemes is consistent across all three types of applications that make different types of decisions, as shown in Table 3. In fact, ChameleonAPI offers the highest accuracy by a clear margin for every single application in our evaluation, as shown in Figure 6.

To better compare the ChameleonAPI approach with Categorized models, we divide the 57 applications into two types: (1) 39 single-category applications — each application uses labels that belong to one category and hence can benefit from one specialized model in the Categorized models scheme; (2) 18 multi-category applications — each application uses labels that belong to multiple categories. For these applications, the Categorized models scheme feeds the API input to multiple specialized models and combines these models' output to form the API output. As shown in Table 4, the Categorized models scheme does offer improvement from Generic models by considering which labels belong to an application's target classes, particularly for single-category applications. However, both ChameleonAPI and ChameleonAPI$_{basic}$ perform better than Categorized models for both single-category and

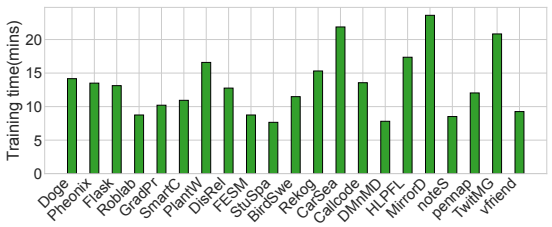


Figure 7: *Re-training time for applications in Figure 6(a).*

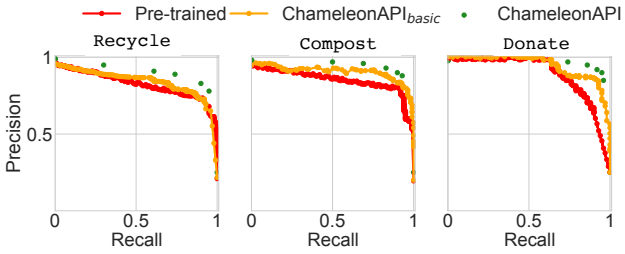


Figure 8: *Precision-recall trade-off for* `HeapsortCypher`.

multi-category applications—the per-application customization in ChameleonAPI and ChameleonAPI$_{basic}$ has paid off.

The above advantage of ChameleonAPI over ChameleonAPI$_{basic}$ and Categorized models shows that the static analysis used in ChameleonAPI to extract not only what labels are used by the application, but also which labels belong to the same target class, the decision type, and the matching order, as described in Section 3.3, is worthwhile.

Cost of obtaining customized models: The customization effort of ChameleonAPI includes two parts (1) extracting the decision-process summary from application source code, and (2) re-training the ML model. The first part takes a few seconds: on an Intel(R) Xeon(R) E5-2667 v4 CPU machine, our parser extracts the decision-process summary from every benchmark application within 10 seconds.

The second part takes a few minutes, much faster than training a neural network from scratch. As shown in Figure 7, re-training DNNs for the 21 applications in Figure 6(a) on a single RTX 3080 GPU takes 8 to 24 minutes. Focusing on a small portion of all possible labels (§2.4), ChameleonAPI fine-tunes pre-trained models using much less training data than the generic models and thus needs fewer iterations to converge.

Considering that a V100 GPU with similar processing GFLOPS as our RTX 3080 GPU only costs $2.38 per hour on Google Cloud [21], re-training an ML model for one application costs less than $1.

Cost of hosting customized models: For cloud providers, ChameleonAPI would incur a higher hosting cost than traditional ML APIs by serving a customized DNN for every application instead of a generic DNN for all applications.

The extra cost includes more disk space to store customized neural network models. For example, each image-classification model in ChameleonAPI uses 115 MB of disk space. So, for n applications, $115 \cdot n$ MB of disk space may be needed to store ChameleonAPI customized models.

The extra cost also involves more GPU resources. A naive design of using one GPU to exclusively serve requests to one customized neural network model will likely lead to under-utilization of GPU resources. To serve different applications' customized models on one GPU, we need to pay attention to memory working set and performance isolation issues. In our experiments on an RTX 3080 GPU, loading an image-classification model from CPU to GPU RAM takes 18 to 40 ms (inference itself takes 10 to 35 ms with a batch size of 1). Fortunately, modern GPU has sufficiently large RAMs to host several requests to different customized models simultaneously: in our experiments, the peak memory consumption of one inference request is less than 2GB. Furthermore, the majority of the model inference memory consumption comes from intermediate states, instead of the model itself. Consequently, the memory consumption of multiple inference requests on different models is similar to that on the same model.

Of course, ChameleonAPI can take advantage of recent proposals to improve GPU sharing [15, 55, 71, 73] as well as to reduce the footprint of serving multiple DNNs [33]. These techniques could be advantageously employed by ChameleonAPI to determine the optimal degree of sharing among customized DNNs, and we leave them to future work.

Finally, there is also the extra cost of needing more complex software to manage the DNN serving. For instance, ChameleonAPI needs to dynamically route each request to a GPU that serves the DNN of the application (see §3.4).

Precision-recall tradeoffs: Traditionally, for a trained ML model, it is common to vary the confidence-score thresholds in order to find the best precision-recall tradeoff of a trained model. Thus, it is important that ChameleonAPI also achieves better precision-recall tradeoffs. Figure 8 shows the precision-recall results in each target class of a particular application, by varying the detection threshold θ (defined in §3.2) of two baselines (real APIs are excluded, because we cannot change their thresholds and their IDR is not as low as ChameleonAPI$_{basic}$). ChameleonAPI's tradeoffs are better than both baselines (and we observe similar results in other applications). Note that since ChameleonAPI's loss function uses an assumed θ, we do not vary the θ when testing it; instead, we re-train five DNNs of ChameleonAPI, each with a different θ and test them with their respective thresholds.

Understanding the improvement: ChameleonAPI's unique advantage is that it factors in the decision process of an application, including not only the target classes but also the decision type and the matching order. Next, we use two case studies to further reveal the underlying tradeoffs made by ChameleonAPI to achieve its improvement on application-decision accuracy.

First, ChameleonAPI reduces errors related to different

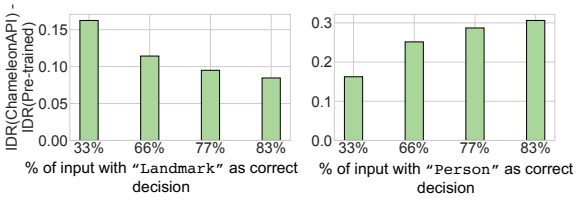


Figure 9: *How the accuracy advantage of ChameleonAPI changes with different input distribution.*

target classes differently depending on their different roles in the application decision process. This effect is particularly striking in Multi-Choice applications with the matching order of App-Order, where the first target class is always matched against API output. Thus, when the correct target class is not the first one, falsely including a label that belongs to the first target class will more likely be a critical error than other mis-classifications, because it will block the match of other target classes. To illustrate this, we consider the Multi-Choice application of `Aander-ETL`. We increase the percentage of testing inputs whose correct action is the first target class or the last target class. Figure 9 shows that increasing the portion of inputs of the last target class (`Person`) generally leads to bigger gains of ChameleonAPI, whereas increasing the portion of the first target class (`Landmark`) does the opposite. This shows the application itself already has good tolerance to mis-classification on inputs that belong to the first target class, but not to mis-classification on the inputs that belong to later target classes, which is exactly where ChameleonAPI can help.

Second, recall from §3.2 that our loss function helps to minimize critical errors, even at the cost of missing labels that do not affect application decisions (*i.e.,* non-critical errors). To show this, we define *label error rate* on an image as the fraction of the image's ground-truth labels that are missed by the DNN output (a label list). We consider `IoTWor` (explained in Table 5), which similar to `Anander-ETL` makes Multi-Choice decisions with App-Order matching order. The average label missing rate of ChameleonAPI on our testing images is 0.21, which is slightly higher than ChameleonAPI$_{basic}$'s 0.18. This means ChameleonAPI makes more label-level mistakes than ChameleonAPI$_{basic}$. However, our IDR (0.17) is 44% lower than ChameleonAPI$_{basic}$, which means ChameleonAPI makes far fewer critical errors.

6 Related Work

Due to space constraints, we discuss related papers that have not been discussed earlier in the paper.

Optimizing storage and throughput of DNN serving: Various techniques have been proposed to optimize the delay, throughput, and storage of ML models via model distillation [40, 54, 61], pruning [26] or cascading [4, 7]. This line of work explores a different design space than ChameleonAPI: they design ML models with higher inference speed or smaller model size with minimum loss in accuracy. ChameleonAPI focuses on re-training existing ML models such that the rate of incorrect decisions of a given application is reduced.

Application-side optimization: Recent work also proposes to change the applications to better leverage existing ML APIs. One line of work [11, 12, 69] invokes ML APIs from different service providers to achieve high accuracy within a query cost budget. Another line of work aims to eliminate misuse of ML APIs in applications [65, 66]. They require changes to the application source code (*e.g.,* changing the API input preparation, switching from image-classification API to object-detectin API, etc.). They are complementary to our work, because we customize the ML-API backend DNN and do not require changes on the application's source code.

Measurement work on MLaaS: For their rising popularity, ML-as-a-Service platforms have also attracted many measurement studies to understand accuracy [10], performance [70], robustness [28], and fairness [41]. However, they have so far not taken in account the ML applications that use ML APIs, and is thus different from our empirical study of ML applications in §2. Previous work that studies ML applications [65] did not look into the decision making process and how ML API errors might affect different applications differently.

Finally, a myriad of techniques have been studied to better manage and schedule GPU resources in ML training/serving systems (*e.g.,* [13, 16, 17, 22, 25, 27, 32, 42, 45, 53, 59, 60, 67, 72, 74]). They aim for different goals than ChameleonAPI, but these techniques can be used to help ChameleonAPI train and serve the application-specific ML models.

7 Conclusion

ML APIs are popular for its accessibility to application developers who do not have the expertise to design and train their own ML models. In this paper, we study how the generic ML models behind ML APIs might affect different applications' control-flow decisions in different ways, and how some ML API output errors may or may not be critical due to the application decision making logic. Guided by this study, we have designed ChameleonAPI that offers both the accuracy advantage of a custom ML model and the accessibility of the traditional ML API.

8 Acknowledgement

We thank all the reviewers for their insightful feedback and suggestions. The project is funded by NSF CNS-2146496, CNS-2131826, CNS-2313190, CNS-1901466, CNS-1956180, CCF-2119184, UChicago CERES Center, and Marian and Stuart Rice Research Award. The project is also supported by Chameleon Projects [38].

References

[1] Aander-ETL. A smart album application. https://github.com/Grusinator/Aander-ETL.

[2] Amazon. Amazon artificial intelligence service. Online document https://aws.amazon.com/machine-learning/ai-services, 2022.

[3] Amazon. Amazon rekognition: detecting labels. Online document https://docs.aws.amazon.com/rekognition/latest/dg/labels.html, 2022.

[4] Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1466–1477. IEEE, 2019.

[5] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[6] Emanuel Ben-Baruch, Tal Ridnik, Nadav Zamir, Asaf Noy, Itamar Friedman, Matan Protter, and Lihi Zelnik-Manor. Asymmetric loss for multi-label classification. *arXiv preprint arXiv:2009.14119*, 2020.

[7] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Thia: Accelerating video analytics using early inference and fine-grained query planning. *arXiv preprint arXiv:2102.08481*, 2021.

[8] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Figo: Fine-grained query optimization in video analytics. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 559–572, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Zhuoqing Chang, Shubo Liu, Xingxing Xiong, Zhaohui Cai, and Guoqing Tu. A survey of recent advances in edge-computing-powered artificial intelligence of things. *IEEE Internet of Things Journal*, 8(18):13849–13875, 2021.

[10] Lingjiao Chen, Tracy Cai, Matei Zaharia, and James Zou. Did the model change? efficiently assessing machine learning api shifts. *arXiv preprint arXiv:2107.14203*, 2021.

[11] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalMCT: Efficient online ML API selection for multi-label classification tasks, 2022.

[12] Lingjiao Chen, Matei Zaharia, and James Y Zou. Frugalml: How to use ml prediction apis more accurately and cheaply. *Advances in neural information processing systems*, 33:10685–10696, 2020.

[13] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022.

[14] COCO. Coco dataset. https://cocodataset.org/#home, 2017.

[15] NVIDIA Corporation. Nvidia multi-process service (mps) documentation, 2023. Accessed: 2023-07-12.

[16] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[19] Amr ElRafey and Janusz Wojtusiak. Recent advances in scaling-down sampling methods in machine learning. *Wiley Interdisciplinary Reviews: Computational Statistics*, 9(6):e1414, 2017.

[20] Google. Google cloud ai. Online document https://cloud.google.com/products/ai, 2022.

[21] Google. Compute engine pricing. Online document https://cloud.google.com/compute/all-pricing, 2023.

[22] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[23] Armin Haller, Axel Polleres, Daniil Dobriy, Nicolas Ferranti, and Sergio J Rodríguez Méndez. An analysis of links in wikidata. In *European Semantic Web Conference*, pages 21–38. Springer, 2022.

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

[25] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[26] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 123–136, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Hossein Hosseini, Baicen Xiao, and Radha Poovendran. Google's cloud vision api is not robust to noise. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pages 101–105. IEEE, 2017.

[29] Haochen Hua, Yutong Li, Tonghe Wang, Nanqing Dong, Wei Li, and Junwei Cao. Edge computing with artificial intelligence: A machine learning perspective. *ACM Comput. Surv.*, aug 2022. Just Accepted.

[30] Zhang Huangzhao. Yahoo_question_answer. https://github.com/LC-John/Yahoo-Answers-Topic-Classification-Dataset.git, 2018.

[31] M Irlbeck et al. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering*, 40:26, 2015.

[32] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 250–263, 2019.

[33] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, Boston, MA, July 2018. USENIX Association.

[34] Philipp Jund, Andreas Eitel, Nichola Abdo, and Wolfram Burgard. Optimization beyond the convolution: Generalizing spatial relations with end-to-end metric learning. *CoRR*, abs/1707.00893, 2017.

[35] Heechul Jung, Sihaeng Lee, Junho Yim, Sunjeong Park, and Junmo Kim. Joint fine-tuning in deep neural networks for facial expression recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 2983–2991, 2015.

[36] Daniel Kang, Peter Bailis, and Matei Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *arXiv preprint arXiv:1805.01046*, 2018.

[37] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *arXiv preprint arXiv:1703.02529*, 2017.

[38] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 219–233. USENIX Association, July 2020.

[39] Phillip Keung, Yichao Lu, György Szarvas, and Noah A. Smith. The multilingual amazon reviews corpus. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 2020.

[40] Hakbin Kim and Dong-Wan Choi. Pool of experts: Real-time querying specialized knowledge in massive neural networks. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2244–2252, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Allison Koenecke, Andrew Nam, Emily Lake, Joe Nudell, Minnie Quartey, Zion Mengesha, Connor Toups, John R Rickford, Dan Jurafsky, and Sharad Goel. Racial disparities in automated speech recognition. *Proceedings of the National Academy of Sciences*, 117(14):7684–7689, 2020.

[42] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: Erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 30–46, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Nick Koudas, Raymond Li, and Ioannis Xarchakos. Video monitoring queries. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[44] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. The open images dataset v4. *International Journal of Computer Vision*, 128(7):1956–1981, 2020.

[45] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, 2018.

[46] Shigang Liu, Jun Zhang, Yang Xiang, Wanlei Zhou, and Dongxi Xiang. A study of data pre-processing techniques for imbalanced biomedical data classification. *International Journal of Bioinformatics Research and Applications*, 16(3):290–318, 2020.

[47] David Marby and Nijiko Yonskai. Pyan3: Offline call graph generator for python 3. https://github.com/davidfraser/pyan.

[48] Petru Martincu. Cloud-computing. https://github.com/Martincu-Petru/Cloud-Computing, 2020.

[49] Chu Matthew. heapsortcypher. https://github.com/matthew-chu/heapsortcypher.git, 2019.

[50] Patrick McEnroe, Shen Wang, and Madhusanka Liyanage. A survey on the convergence of edge computing and ai for uavs: Opportunities and challenges. *IEEE Internet of Things Journal*, 9(17):15435–15459, 2022.

[51] Microsoft. Microsoft azure cognitive services. Online document https://azure.microsoft.com/en-us/services/cognitive-services, 2022.

[52] Microsoft. Microsoft azure image tagging. Online document https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/concept-tagging-images, 2022.

[53] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

[54] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, and Kayvon Fatahalian. Online model distillation for efficient video inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3573–3582, 2019.

[55] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for Memory-Efficient, Real-Time video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 973–994, Boston, MA, April 2023. USENIX Association.

[56] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[57] Plant-Watcher. A plant management application. https://github.com/siwasu17/plant-watcher.

[58] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[59] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[60] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[61] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3646–3654, 2017.

[62] The-Coding-Kid. A smart album application. https://github.com/The-Coding-Kid/888hacks-flask.

[63] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147, 2003.

[64] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171:107138, 2020.

[65] Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Are machine learning cloud apis used correctly? In *43th International Conference on Software Engineering (ICSE'21)*, 2021.

[66] Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Automated testing of software that uses machine learning apis. In *44th International Conference on Software Engineering (ICSE'22)*, 2022.

[67] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

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

[69] Shuzhao Xie, Yuan Xue, Yifei Zhu, and Zhi Wang. Cost effective mlaas federation: A combinatorial reinforcement learning approach. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2022.

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

[71] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.

[72] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[73] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[74] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. {DeepCPU}: Serving {RNN-based} deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, 2018.

Appendix A Applications

Table 5: The statistics of 77 applications in empirical study. (Multi-Choice-* refer to Multi-Choice (*-Order).)

Application name (Link to Github repo)	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
Image Multi-Label Classification (Google `label_detection`, AWS `detect_labels`)			
2019-iot-ai-workshop	Multi-Choice-App	**2** (7, 2)	[Capuchin monkey, ...]; [Wildlife biologist, ...]
Aander-ETL	Multi-Choice-App	**3** (9, 6, 5)	[Landmark, Sculpture, ...]; [Building, Estate, ...]; [Human, ...]
ArtGuide	Multi-Choice-API	**2** (6, 3)	[Painting, Picture frame, ...]; [Building, Architecture, ...]
AWS_CloudComputing	Multi-Select	**2** (1, 1)	[Hot dog]; [Food]
DoorWatch	True-False	**1** (6)	[Clothing, Person, Human, Furniture, Child, Man]
AWSRekognition	Multi-Select	**2** (3, 3)	[Person, People, Human]; [Art, Drawing, Sketch]
GraduateProject	True-False	**1** (5)	[Orator, Professor, Projection Screen, ...]
Voice-Assistant	Multi-Select	**3** (5, 3, 1)	[Highway, Lane, ...]; [Car, ...]; [Classroom]
callforcode	True-False	**1** (5)	[Water, Waste, Bottle, Plastic, Pollution]
Car-Image-search	True-False	**1** (6)	[Sedan, Mini SUV, Coupe utility, Truck, Van, Convertible]
cloudComputing_project2	Multi-Choice-API	**3** (1, 3, 1)	[Person]; [Dog, Cat, Mammal]; [Flower]
CSC847_GAE_Proj2	Multi-Choice-API	**3** (2, 2, 1)	[Mammal, Livestock]; [Human, People]; [Flower]
cutiehack	Multi-Select	**2** (1, 3)	[Banana]; [Lemon, Citrus fruit, Apple]
CycleGAN-tensorflow_pixie	Multi-Choice-API	**3** (1, 6, 4)	[Food]; [Girl, Boy, Man, ...]; [Room, Living room, House, ...]
DisasterRelief	True-False	**1** (8)	[Hurricane, Flood, Tornado, Landslide, Earthquake, Volcano, ...]
Dogecoin_musk	True-False	**1** (4)	[Dog, Mammal, Carnivore, Wolf]
flaskAPI	True-False	**1** (3)	[Food, Recipe, Ingredient]
food-assessment-system	Multi-Choice-API	**5** (35, 22, 54, 4, 6)	[Dessert, ...]; [Grilling, ...]; [Strawberries, ...]; [Cigarette, ...]; ...
Foodier	Multi-Select	**2** (13, 1)	[Building, Logo, Menu, Person, Vehicle, People, ...]; [Food]
Hack-At-Home-II	Multi-Choice-API	**2** (3, 3)	[Food, Junk food, Plastic]; [Drinkware, Wood, Metal]
HeapSortCypher	Multi-Choice-API	**3** (8, 5, 11)	[Food, Food grain, ...]; [Clothing, Shirt, ...]; [Paper bag, ...]
IngredientPrediction	Multi-Select	**3** (1, 1, 1)	[Spaghetti]; [Bean]; [Naan]
FESMKMITL	True-False	**1** (1)	[Face]
milab	Multi-Choice-App	**3** (1, 1, 1)	[Sign]; [Nature]; [Car]
BirdSwe	Multi-Choice-API	**1** (5)	[Smoke, Bird, ...]
ai-server-proto	Multi-Choice-API	**3** (3, 14)	[Eye, Eyeball, Eyes]; [Landmark, Sculpture, Monument, ...]
Pheonix	True-False	**1** (1)	[Fire]
photo_book	Multi-Choice-API	**3** (10, 10, 2)	[Mammal, Bird, Insect, ...]; [Skin, Lip, ...]; [Flower, Plant]
Plant-Watcher	True-False	**1** (5)	[Plant, Flowerpot, Houseplant, Bonsai, Wood]
RecBot	Multi-Choice-App	**2** (11, 8)	[Tin, Paper, Magazine, Carton, ...]; [Food, Bread, Pizza, ...]
roblab-hslu	True-False	**1** (7)	[Raincoat, Coat, Jacket, T-shirt, Trousers, Jeans, Shorts]
senior-project	Multi-Choice-API	**3** (2, 3, 1)	[Landscape, Landmark]; [Self-portrait, Portrait, ...]; [Flower]
smart-can	True-False	**1** (9)	[Paper, Bottle, Plastic, Container, Tin can, Glass, ...]
smart-trash-bin	Multi-Choice-API	**2** (14, 5)	[Aviator sunglass, Beer glass, ...]; [Plastic arts, ...]
smartHamper	Multi-Choice-API	**3** (7, 4, 3)	[Shirt, T-shirt, ...]; [Trousers, Denim, ...]; [Brand, Text, ...]
StudySpaceAvailability	True-False	**1** (4)	[Hardware, Power Drill, Drill, Electronics]
The-Coding-Kid	Multi-Select	**6** (9, 6, 9, 3, 6, 3)	[Noodle, ...]; [Meat, ...]; [Produce, ...]; [Fruit, ...]; [Milk, ...]; ...
Tinyml	Multi-Select	**3** (4, 6, 5)	[Car, Truck, ...]; [Gun, Weapon Violence, ...]; [Cat, Dog, ...]
UofTHacksBackend	Multi-Choice-API	**4** (3, 3, 7, 4)	[T-shirt, ...]; [Outerwear, ...]; [Pants, ...]; [Footwear, ...]
garbage-sort	Multi-Choice-API	**2** (1, 20)	[Food];[Metal, ...];
Image Object Detection (Google `object_localization`)			
equipment-detection-poc	Multi-Select	**1** (1)	[Shoe]
flood_depths	Multi-Select	**1** (5)	[Car, Van, Truck, Boat, Toy vehicle]
SBHacks2021	Multi-Select	**1** (1)	[Person]
SeeFarBeyond	Multi-Select	**1** (2)	[Spoon, Coin]

(To be continued)

Table 5: The statistics of 77 applications in empirical study (Continued).

Application	Decision Type (Matching Order)	# of Target Classes (# of labels per class)	Branch Conditions (Class lists or value ranges are separated by semicolons.)
shecodes-hack	Multi-Select	**1** (2)	[Dress, Top]
SunHacks2019	Multi-Select	**1** (3)	[Person, Chair, Table]
thgml	Multi-Select	**1** (7)	[Pizza, Food, Sushi, Baked goods, Snack, Cake, Dessert]
Verlan	Multi-Select	**1** (2)	[Dog, Animal]
Text Sentiment Classification (Google `sentiment_detection`)			
animal-analysis	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.5, 1]; [0, 0.5]; [-0.5, 0]; [-1, -0.5]
calhacksv2	Multi-Choice-API	**6** (1, 1, 1, 1, 1, 1)	[0.5, 1]; [0.5, 1]; [0.1, 0.5]; [-0.1, 0.1]; [-0.5, -0.1]; [-1, -0.5]
carbon_hack_sentiment	Multi-Choice-API	**3** (1, 1, 1)	[0.3333, 1]; [-0.3333, 0.3333]; [-1, -0.3333]
FoodDelivery	Multi-Choice-API	**3** (1, 1, 1)	[0.6, 1]; [0.3, 0.6]; [-1, 0.3]
devfest	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.6, 1]; [0.4, 0.6]; [0.2, 0.4]; [-1, 0.2]
EC601_twitter_keyword	Multi-Choice-API	**3** (1, 1, 1)	[0.25, 1]; [-0.25, 0.25]; [0.25, 1]
ElectionSentimentAnalysis	Multi-Choice-API	**3** (1, 1, 1)	[0.05, 1]; [0, 0.05]; [-1, 0]
Hapi	Multi-Choice-API	**2** (1, 1)	[-1, 0]; [0, 1]
JournalBot	Multi-Choice-API	**3** (1, 1, 1)	[0.5, 1]; [0, 0.5]; [-1, 0]
Mind_Reading_Journal	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.15, 1]; [0.1, 0.15]; [-0.15, 0.1]; [-1, -0.15]
Sarcatchtic-MakeSPP19	Multi-Choice-API	**2** (1, 1)	[-0.5, 1]; [-1, -0.5]
stockmine	Multi-Choice-API	**2** (1, 1)	[-1, 0]; [0, 1]
Tone	Multi-Choice-API	**3** (1, 1, 1)	[-1, -0.5]; [-0.5, 0.5]; [0.5, 1]
UOttaHack_2019	Multi-Choice-API	**3** (1, 1, 1)	[0.25, 1]; [-0.25, 0.25]; [-1, -0.25]
Text Entity Detection (Google `entity_detection`)			
GeoScholar	True-False	**1** (1)	[LOC]
HackThe6ix	Multi-Choice-API	**7** (1, 1, 1, 1, 1, 1, 1)	[PERSON]; [LOC]; [ADD]; [NUM]; [DATE]; [PRICE]; [ORG]
Klassroom	Multi-Choice-API	**2** (2, 2)	[PERSON, PROPER]; [LOC, ORG]
newsChronicle	True-False	**1** (1)	[OTHER]
ocr-contratos	True-False	**1** (1)	[NUM]
uofthacks6	True-False	**1** (1)	[OTHER]
Text Topic Classification (Google `text_classify`)			
DMnMD	True-False	**1** (1)	[Health]
HLPFL	True-False	**1** (8)	[Public Safety, Law & Government, Emergency Services, News, ...]
MirrorDashboard	True-False	**1** (7)	[Jobs & Education, Law & Government, News, ...]
noteScript	True-False	**1** (1)	[Food]
pennapps_2019f	True-False	**1** (2)	[News/Politics, Investing]
soap	Multi-Choice-API	**2** (2, 2)	[Sensitive Subjects, ...]; [Discrimination & Identity Relations, ...]
SocialEyes	Multi-Choice-API	**2** (2, 1)	[people & society, sensitive subjects]; [adult]
Twitter_Mining_GAE	True-False	**1** (1)	[Sentitive]
vfriendo	True-False	**1** (1)	[Restaurants]

Appendix B Loss function for other decision-process summaries

True-False:

$$L(y) = \overbrace{\text{Sigmoid}(\max_{l\in G_{\hat{c}}}(\mathbf{y}) - \theta)}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} + \overbrace{\text{Sigmoid}(\theta - \max_{l\in G_c}(\mathbf{y}))}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} \quad (4)$$

Multi-Select:

$$L(y) = \overbrace{\sum_{c\in\hat{T}} \text{Sigmoid}(\theta - \max_{l\in Gc}\mathbf{y}[l])}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} + \overbrace{\sum_{c\in\cup_c G_c\setminus\hat{T}} \text{Sigmoid}(\max_{l\in Gc}\mathbf{y}[l] - \theta)}^{\text{Penalize } \textbf{Type-3} \text{ Critical Errors}} \quad (5)$$

Multi-Choice API-order: Here we explain why this loss function captures the critical errors:

- A Type-1 error occurs, if (1) the correct target class is matched, thus at least one of its labels has a score above the confidence threshold ($\max_{l\in G_{\hat{c}}}\mathbf{y}[l] \geq \theta$), and (2) it is matched after the EOD because all of the labels belonging to the correct target class have scores below the maximum score of labels in the incorrect target classes.
- A Type-2 error occurs if the maximum score for labels in a correct target class falls below threshold θ, thus it is never matched (before or after EOD).
- A Type-3 error occurs if any labels belonging ($\max_{l\notin G_{\hat{c}}}\mathbf{y}[l]$) to incorrect target classes appears before labels in the correct target class.

$$L(\mathbf{y}) = \overbrace{\text{Sigmoid}\left(\max_{l\in\cup_{c\neq\hat{c}}G_c}\mathbf{y}[l] - \max_{l\in G_{\hat{c}}}\mathbf{y}[l]\right)}^{\textbf{Type-1} \text{ Critical Errors}} + \overbrace{\text{Sigmoid}\left(\max_{l\in\cup_{c\neq\hat{c}}G_c}\mathbf{y}[l] - \theta\right)}^{\textbf{Type-2} \text{ Critical Errors}} + \overbrace{\sum_{c\neq\hat{c}}\text{Sigmoid}\left(\max_{l\in G_c}\mathbf{y}[l] - \max_{l\in G_{\hat{c}}}\mathbf{y}[l]\right)}^{\textbf{Type-3} \text{ Critical Errors}} \quad (6)$$

Value ranges: As for APIs that output a score $\mathbf{y}$ to describe the input, applications typically define several value ranges as target classes to make decisions, where the lower bound of the c^{th} target class is denoted as l_c and the upper bound of the c^{th} target class is denoted as u_c.

$$L(y_i) = \overbrace{\text{Sigmoid}(\mathbf{y} - u_{\hat{c}}) + \text{Sigmoid}(l_{\hat{c}} - \mathbf{y})}^{\textbf{Type-1} \text{ Critical Errors}} + \overbrace{\sum_{c\neq\hat{c}}\text{Sigmoid}(u_c - \mathbf{y}) + \text{Sigmoid}(\mathbf{y} - l_c)}^{\textbf{Type-3} \text{ Critical Errors}} \quad (7)$$

where $\hat{c}$ is the index of the correct target class. A Type-1 error occurs (*i.e.,* a correct target class is matched after EOD) when the output score $\mathbf{y}$ exceeds the upper bound of the ground-truth value range (u_c), or falls below the lower bound of the ground-truth value range (l_c). A Type-3 error occurs when the upper bound of an incorrect value range exceeds $\mathbf{y}$ *and* its lower bound falls below $\mathbf{y}$, leading it to be selected. Type-2 errors are absent in this application because all the target classes span the whole output range, thus a target class must be matched.

Appendix C Setup of Categorized models

Here, we describe in detail how we construct the label categories to support the scheme of Categorized models, which is one of the schemes in comparison with ChameleonAPI (Section 5).

Image-classification Image-classification APIs typically contains many thousands of labels without providing their categorization or hierarchy. Therefore, we create categories leveraging the Wikidata knowledge graph [68], a widely used knowledge graph database that has been referred to during the creation of many popular ML training datasets [23, 34, 44]. In this knowledge graph, each node is a named entity, covering *all* the labels used in popular image-classification APIs [2, 20, 52], and each edge represents a relationship between two entities (e.g., "subclass of", "different from", "said to be the same as").

Extracting "subclass of" edges in Wikidata knowledge graph, we get a directed acyclic graph (DAG) of label hierarchy. We believe it offers a principled foundation to create label categories based on two observations: (1) If an entity/node e is reachable from another entity/node e' through several subclass-of edges, e' is also covered by the *category* of e (e.g., entity "motor vehicle" is directly connected to "land vehicle", entity "land vehicle" is directly connected to "vehicle", so "motor vehicle" is also covered by the "vehicle" category); (2) The distance, measured in the number of subclass-of edges, between an entity/node and the DAG root indicates the specificity of the concept behind this node, with shorter distance representing coarser-grained categories. We will refer to a node that is k edges away from the root as a Level-k node.

Based on these observations, we formally define a set of categories C_k for all the image-classification labels L at a specificity level k as follows: C_k is the minimum set of Level-k nodes such that every label $l \in L$ is covered by at least one category node $c_k \in C_k$. We could categorize all the applications into single-category and multi-category applications using any level of specificity settings. In this paper, we adopt Level-2 specificity setting, since the number of single-category applications drops a lot when moving from Level-2 to Level-3, indicating Level-3 categories may be too fine-grained.

Under Level-2, we set up 35 categorized models that cover all the image-classification labels. Six of them are used by applications in our benchmark, including *natural object*, *temporal entity*, *artificial entity*, *system*, *phenomenon*, and *continuant*. With this categorization, 27 of the 40 image-classification applications are single-category, and the rest 13 applications are multi-category.

Object-detection Similar as image-classification API, every object-detection API label also corresponds to an entity node in the Wikidata knowledge graph. Therefore, we use the same methodology and the same specificity Level-2 to define categories for object detection labels.

Seven categorized models are set up to cover all object-detection labels. The object-detection labels used by 8 object-detection benchmark applications belong to 3 categories: *natural object*, *artificial entity*, and *system*. Under this setting, there will be 7 single-category applications, and 1 multi-category applications.

Text-classification The Google text-classification API [20] offers the hierarchy tree of all its labels. We simply follow their categorization and get 15 categories to covering all text-classification labels. Nine categories are used by applications in our benchmark, including *business & industrial*, *people & society*, *health*, *food & drink*, *jobs & education*, *news*, *sensitive subjects*, *adult*, and *law & government*. Under this categorization, there are 5 single-category text-classification applications, and 4 multi-category text-classification applications.

Other types of applications There are two other types of applications in our benchmark that are not suitable for designing pre-specialized models: sentiment analysis and named entity recognition.

For sentiment analysis API, the corresponding applications typically define several value-ranges and determine which range the API output (a sentiment score) falls into. Since the API output is a floating point number, there are infinite ways of defining value-ranges. Therefore, it is impracticable to create pre-categorized models.

For named entity recognition API, it only has 6 labels: *person, location, organization, number, date*, and *misc* [20]. They are already high-level categories. There is no need to create pre-categorized models for each category.

Appendix D Results of other applications

As mentioned in §5.1, the results of the 20 applications that involve sentiment analysis and entity recognition were not included in the evaluation section. Their results are shown in Figure 10 and 11. As we can see, the advantage of ChameleonAPI is consistent across these applications, similar to what we presented in§5. Note that, the scheme of Categorized models does not apply to applications that involve these two types of ML tasks, and hence is not included in Figure 10 and 11.

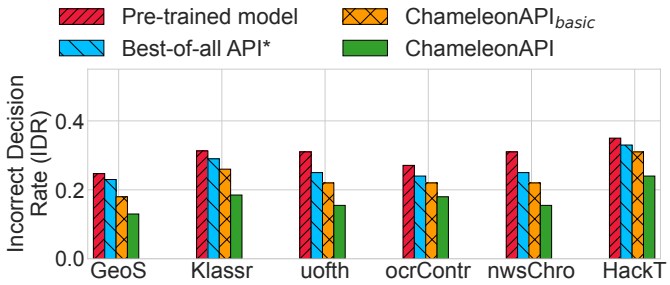


Figure 10: Results on entity-recognition applications.

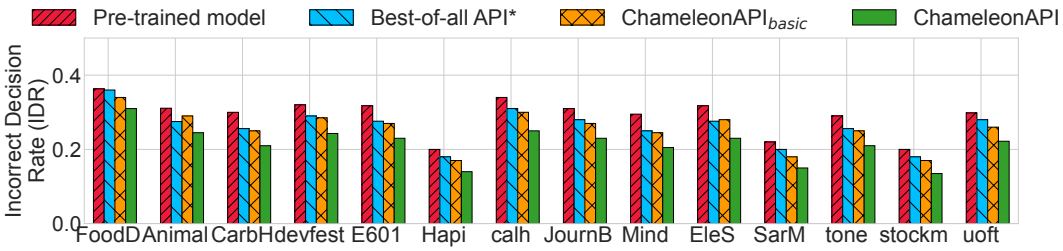


Figure 11: Results on sentiment-analysis applications.