AutoConsis: Automatic GUI-driven Data Inconsistency Detection of Mobile Apps

Yongxiang Hu*
School of Computer Science
Fudan University
Shanghai, China

Hailiang Jin
Meituan
Shanghai, China

Xuan Wang*
School of Computer Science
Fudan University
Shanghai, China

Jiazhen Gu
School of Computer Science
Fudan University
Shanghai, China

Shiyu Guo
Meituan
Shanghai, China

Chaoyi Chen
Meituan
Beijing, China

Xin Wang
School of Computer Science
Fudan University
Shanghai Key Laboratory of
Intelligent Information Processing
Shanghai, China

Yangfan Zhou
School of Computer Science
Fudan University
Shanghai Key Laboratory of
Intelligent Information Processing
Shanghai, China

ABSTRACT

In industrial practice, many bugs in commercial mobile apps manifest as self-conflicts of data presented in the GUI (Graphical User Interface). Such *data inconsistency bugs* can bring confusion to the users and deteriorate user experiences. They are a major target of industrial testing practice. However, due to the complication and diversity of GUI implementation and data presentation (*e.g.*, the ways to present the data in natural language), detecting data inconsistency bugs is a very challenging task. It still largely relies on manual efforts. To reduce such human efforts, we proposed AutoConsis, an automated data inconsistency testing tool we designed for Meituan. one of the largest E-commerce providers with over 600 million transacting users. AutoConsis can automatically analyze GUI pages via a multi-modal deep-learning model and extract target data from textual phrases leveraging LLMs (Large Language Models). With these extracted data, their inconsistencies can then be detected. We evaluate the design of AutoConsis via a set of ablation experiments. Moreover, we demonstrate the effectiveness of AutoConsis when applying it to real-world commercial mobile apps with eight representative cases.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software evolution.*

KEYWORDS

Automatic Testing, Mobile Apps, Functional Bug, In-context Learning

ACM Reference Format:
Yongxiang Hu, Hailiang Jin, Xuan Wang, Jiazhen Gu, Shiyu Guo, Chaoyi Chen, Xin Wang, and Yangfan Zhou. 2024. AutoConsis: Automatic GUI-driven Data Inconsistency Detection of Mobile Apps. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3639477.3639748

*This work was performed during the authors' internship in Meituan.

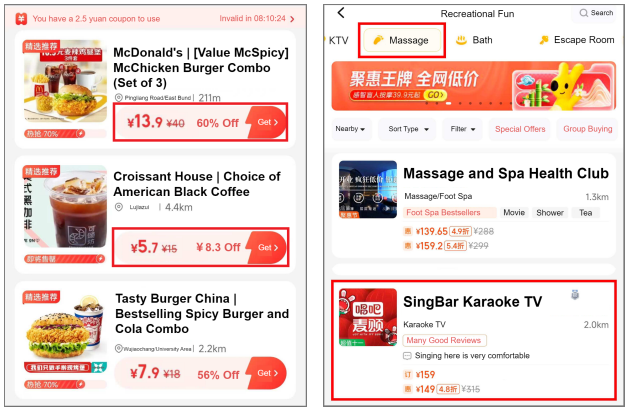
ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0501-4/24/04…$15.00
https://doi.org/10.1145/3639477.3639748

1 INTRODUCTION

Online commercial apps (*e.g.*, those for online shopping) are typical with multiple complicated functions. Bugs in such apps are inevitable given its complication [23]. In our quality assurance experiences in Meituan [5], one of the largest E-commerce providers with over 600 million users [6], we find that many bugs manifest themselves as self-conflicts of data presented in the GUI (Graphical User Interface). Figure 1 presents two cases of such bugs. Such *data inconsistency bugs* usually can bring confusion to the users and hence deteriorate user experience. It is quite necessary to detect such bugs in-house before the target apps are released to the end-users.

Unfortunately, realizing a practical, effective solution to perform data inconsistency detection in the GUI is a very challenging task. Specifically, every business functions in a commercial app typically have diverse GUI implementations and results in diverse look-and-feel GUI pages, where a page is one GUI screen. For example, the advertisement (AD) pages feature various design patterns (*e.g.*, a poster for a single product, or a product list on Sale). Moreover, data on these GUI pages are typically represented in textual phrases with diverse natural language expressions. Extracting data relies on understanding of the natural language phrases inevitably. For



(a) AD page with wrong price calculation (b) A Karaoke appear in the Massage category

Figure 1: Data inconsistency bugs

example, the discount data "$8.3 Off" and "60% Off" are both possible valid expressions, as shown in Figure 1, which should be consistent.

Although automated testing has long been advocated as one critical approach for mobile apps' bug detection [31, 40], current approaches are still not mature in detecting data inconsistency bugs. For example, automated API testing tools [26] can efficiently validate the implementations of APIs. However, incorrect invocations of APIs or mistakes in GUI codes per se can still result in data inconsistencies on GUI pages. There is also a rich literature on automated testing tools for mobile apps [7, 13, 17, 34, 41]. However, they generally do not focus on detecting data inconsistencies. Instead, their design is typically on non-functional bug detection (*e.g.*, crashes).

In industry practices, detecting data inconsistency bugs still largely relies on intensive manual efforts. For example, data inconsistency detection in Meituan is performed mainly through GUI script testing. Specifically, testing engineers should manually select various textual phrases containing the target data by XPath, a language for pointing to different parts from a UI hierarchy file [8]. Then, multiple rules are designed to extract the target data from the textual phrases. Finally, testing engineers require to specify the relationship among these target data with the test script. However, commercial mobile apps typically have multiple business lines (*e.g.*, Hotel booking, Taxi Booking and Food Delivery). Each of these lines involves tremendous pages. Detecting possible data inconsistency bugs requires huge manual efforts in developing test scripts accordingly. This paper aims to reduce such manual efforts in combating data inconsistency bugs.

We can find that manual efforts are mostly on GUI page analysis and natural language phrase understanding. Hence, the key to reduce such human efforts is to design automatic approaches on these tasks. In this regard, we design AutoConsis, an automatic data inconsistency bug detection tool for commercial mobile apps. Unlike existing GUI testing scripts, AutoConsis follows the way humans perform data recognition from GUI pages. Specifically, by formulating the GUI page analysis as an object detection task, AutoConsis recognizes GUI sub-regions containing target data via a specially tailored multi-modal CLIP [30] model. For each GUI sub-region, in order to extract target data from the textual phrases, AutoConsis converts the human understanding procedure into a chain-of-thought (COT) prompt [42]. Then the target data extraction can be automatically conducted by large language models (LLMs).

We analyze the design effectiveness of AutoConsis, with a set of ablation experiments. We prove that AutoConsis design performs the best in both GUI sub-region recognition and data extraction procedures. We apply AutoConsis to practical commercial mobile apps. Our experimental results show that AutoConsis can effectively extract target data from GUI pages, and can recall 93% data inconsistency bugs while maintaining a false positive rate of less than 5%. We also report our field experiences in applying AutoConsis in Meituan. We present ten real-world data inconsistency bug cases. We summarize the contributions of this paper as follows.

- We introduce and analyze data inconsistency bugs, an important category of functional problems of commercial mobile apps.
- We propose AutoConsis, an automated data inconsistency testing tool. AutoConsis adopts a specifically-tailored learning algorithm to model human perceptions, so as to check data correlations for the target apps.
- We show that AutoConsis is an effective tool via real-world case studies on commercial apps that serve tremendous end-users. We summarize the lessons learned, which can shed light on the automation of functional testing.

The rest of the paper is organized as follows. Section 2 introduces practices of mobile app testing and challenges in decreasing manual efforts. In Section 3, we demonstrate the design of AutoConsis by proposing each of its three parts. The experiment is presented in Section 4. Section 5 elaborates our practical cases for the deployment of AutoConsis. Finally, the paper is concluded in Section 7.

2 BACKGROUND AND CHALLENGES

In this section, we first introduce current practices in Meituan in detecting data inconsistency bugs. Upon examining the development of GUI testing scripts, we figured out the two procedures in data inconsistency detection that demand the most manual effort. Finally, we demonstrate the challenges in automating these two procedures.

2.1 Practices of Mobile App Testing

The design of AutoConsis is motivated by the practical testing requirements of Meituan's mobile app. Meituan is one of the largest online shopping platforms over the world, with nearly 700 million transacting users and about 10 million active merchants. Data inconsistency bugs in such apps would inevitably deteriorate user experience.

In order to detect data inconsistency bugs, testing engineers in Meituan currently adopt two types of testing approaches, *i.e.*, automated API testing [19, 19, 26] and GUI scripts testing [17]. Automated API testing is designed for server-side interfaces (*e.g.*, those to access product database). However, since GUI serves as the medium through which users interact, the consistency of APIs

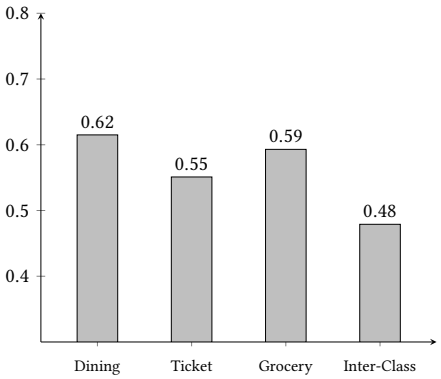


Figure 2: Visual similarity of different GUI pages

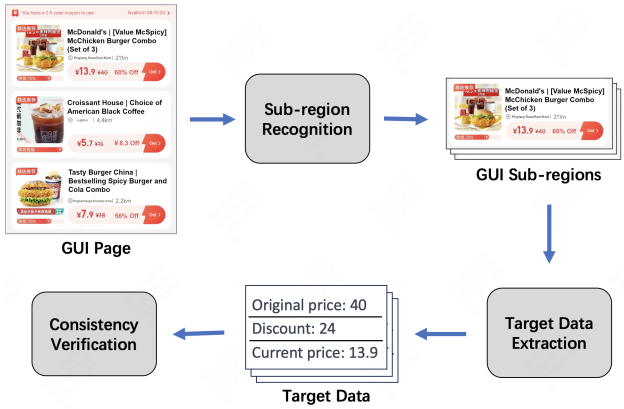


Figure 3: Overview of AutoConsis

cannot guarantee the consistency of the GUI pages. Therefore, even if all the automated API testing is successfully passed, users may still experience data inconsistency bugs.

Although GUI script testing can identify data inconsistency bugs from the user's perspective, the labor-intensive script development limits its testing coverage, making it insufficient for comprehensive data inconsistency testing. For example, GUI testing scripts for Meituan's mobile apps have to be hard-coded. Testing engineers developing rules to select target data from the UI hierarchy [4] and perform consistency verification.

Given that each GUI page has a unique UI hierarchy structure, testing engineers have to develop different GUI testing scripts for each page to detect data inconsistencies. However, Meituan's mobile apps have many different business lines (*e.g.*, Hotel booking, Taxi Booking and Food Delivery), and each line has thousands of goods and serves hundreds of cities. Therefore, achieving comprehensive coverage of data inconsistency bugs across these pages inevitably requires to develop tremendous GUI testing scripts. Since each script requires an average of 30-40 minutes for a testing engineer to develop, the tremendous number of testing scripts would incur daunting manual efforts. Consequently, reducing the manual effort involved in data inconsistency detection has become a pressing issue.

2.2 Challenges

By analyzing the procedure of GUI script development, we found that most of the manual efforts are spent on GUI page analysis and natural language understanding. Therefore, the automation of these two procedures will result in a significant reduction of manual effort in data inconsistency detection. However, there are challenges in the automation of each procedure.

2.2.1 Challenges from GUI Diversity. Since different GUI implementations will result in different analysis procedures, the GUI diversity poses challenges for the automation of GUI analysis.

We conducted an empirical study to further understand the diversity of GUI pages from Meituan's app. Specifically, 100 AD pages were collected from three main business lines in Meituan's commercial app (*i.e.*, Dining Voucher, Admission Ticket and Grocery Shopping). For each of the two AD pages, we leveraged the widely used structural similarity index measure (SSIM) to represent the visual similarity [15].

Figure 2 shows the visual similarity among different GUI pages. We can find that, on average, the visual similarity between different pages within the same business is 59% and the pages' visual similarity between different businesses is less than 48%. The diversity of GUI pages in commercial apps poses significant challenges for automatic GUI analysis.

2.2.2 Challenges from Natural Language. Since GUIs are designed for human understanding and interaction, most of the textual information on GUI pages is presented in natural language and comes in various expressions. Therefore, automatically extracting target data from such textual information presents a significant challenge.

For example, as shown in Figure 1(a), the target data *Discount Value* manifests in multiple expressions. In most expressions, it isn't explicitly stated but rather implicitly embedded within the semantic information of textual phrases. Furthermore, mathematical calculations are required to get the *Discount Value* in some expressions.

In Section 3, we will introduce our automatic inconsistency detection tool, AutoConsis, which can effectively extract target data among diverse GUI pages and perform automatic data inconsistency detection.

3 TOOL DESIGN AND TESTING WORKFLOW

3.1 Overview

Figure 3 shows the overview of AutoConsis, a tool designed for automatic data inconsistency detection. AutoConsis comprises three different parts, the GUI sub-region recognition, the target data extraction and the consistency verification. Specifically, AutoConsis first adopts a learning-based object detection approach to recognize sub-regions containing target data from GUI pages. Then, by mimicking the data recognition process of human uses, AutoConsis constructs a COT prompt. Finally, with LLM extracting target data from the natural language phrases, AutoConsis can automatically perform consistency verification.

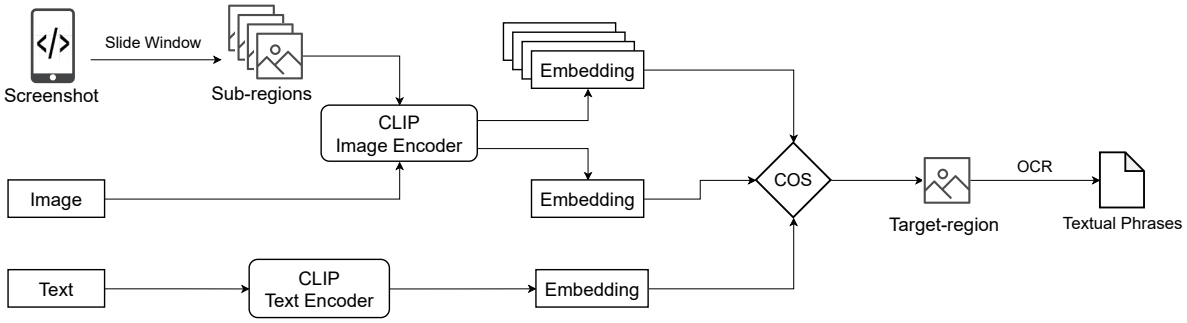


Figure 4: Workflow of the GUI sub-region recognition part of AutoConsis

3.2 GUI Sub-region Recognition

Unlike common domain-specific object detection tasks, as discussed in Section 2.2.1, the diversity across different GUI pages is substantial. Therefore, with the aim of adapting to diverse GUI pages, AutoConsis requires a generic object detection model. CLIP (Contrastive Language-Image Pre-training) calculates the cosine similarity between text and image embeddings to achieve image classification [30]. Due to its fundamental nature as a Language-Image mapping model, it can also be applied to object detection jobs [37]. Moreover, having been trained on huge amounts of (image, text) pairs, CLIP demonstrates excellent generalizability across various scenarios. Therefore, we choose CLIP as the foundational model for the GUI pre-processing component.

Unfortunately, for commercial mobile apps, many GUI sub-regions are difficult to precisely describe with a brief textual phrase (*e.g.*, Product Card in Figure 1(b)). This makes it challenging for pre-trained CLIP models to handle this task directly. However, we've observed that human users can effortlessly identify these GUI sub-regions by leveraging both image and textual information.

Therefore, inspired by the multi-modal recognition pattern from human users, AutoConsis adopted a modified CLIP model to include an additional image input dimension. As shown in Figure 4, for any given GUI screenshot, AutoConsis allows both a textual query (T) and an image (V) to serve as combined search criteria.

Specifically, AutoConsis generates sub-regions $S = \{s_1, s_2, \ldots, s_n\}$ for the GUI screenshot under-recognition through a sliding window. Each sub-region s_i is processed through the image encoder of the CLIP model to obtain its corresponding embedding E_i. Meanwhile, an image V is encoded into E_v using the same encoder and a text segment T is encoded into E_t through the text encoder of CLIP. Since the CLIP model can transfer both text and images to a similar embedding space [30], AutoConsis identifies the target sub-region $Target_{multi-modal}$ by calculating the cosine similarity between these embeddings.

$$k = \arg\max_{i=0}^{n} (\cos(E_i, E_t) + \cos(E_i, E_v)) \quad (1)$$

$$Target_{multi-modal} = s_k \quad (2)$$

After $Tatget_{multi-modal}$ is detected, most of the noise in the screenshot is eliminated. AutoConsis then employs the OCR service from Meituan to obtain **textual phrases** (with target data in it) from $Target_{multi-modal}$.

Example Prompt:

```
Here is a list of phrases from commercial mobile apps:
['Peace Road', 'Garden Cake', 'vanilla cake',
 '66% Sold', 'Dessert', 'Direct subsidy', '45min left',
 '$50', '$30', '$20', '60% off']

Q: Given a text list from commercial mobile apps,
   analyze and extract the following data:

- original price:
- discount:
- price after discount:

A: price-related phrases are ['$50', '$30', '$20',
   '60% off']. '$50' is likely the original price
   as it is the highest of these mentioned. According
   to '60% off' which indicates the discount is '$50'
   *'60%' = '$30', the after discount price is '$50'-
   '$30'='$20'. Therefore, the three target data are
   original price: "50", discount: "30", price after
   discount: "20"
```

Target Prompt: +

```
Here is a list of phrases from commercial mobile apps:
[$Phrases]

Q: Given a text list from commercial mobile apps,
   analyze and extract the following data:

- original price:
- discount:
- price after discount:

A: [To be completed]
```

Figure 5: Chain-of-thought prompt example

3.3 Target Data Extraction

After the GUI sub-regions have been recognized, AutoConsis should extract the target data from the corresponding textual phrases. As demonstrated in Section 2.2.2, the automation understanding of natural language is a challenging task. Inspired by human behavior, and the breakthrough of LLMs [29, 32], AutoConsis conducts target data extraction with the help of LLM.

Specifically, given the testing requirement, test engineers should first define the **target data** that is related to the requirement. Then, AutoConsis takes the **textual phrases** from Section 3.2 and the **target data** as input. By utilizing the prompt template, AutoConsis guides the LLMs to extract the target data from the textual phrases accurately.

3.3.1 Chain-of-Thought Prompt Template. Data inconsistency bugs are often logic-related and therefore the detection procedure typically requires logical inference and calculation of target data from multiple pages. As for the price inconsistency issue in Figure 1(a), the identification of different pricing-coupons modes requires interpretive logic, while the verification of prices within each case demands computational capabilities.

Due to the logical inference demand, AutoConsis leverages the computational and interpretive capabilities of LLMs. Specifically, AutoConsis converts the inconsistency testing requirement into chain-of-thought (COT) prompts [42] to present how to conduct inconsistency analysis. The example prompt in Figure 5 shows the corresponding COT prompt for the food packages in Figure 1(a). Specifically, the COT prompt first guides the LLM to extract all phrases related to the target from the textual phrases. Subsequently, based on the testing requirement, the COT prompt directs the LLM to make better distinctions.

3.3.2 In-context learning. Although having been trained on vast amounts of data, LLMs lack domain knowledge for commercial app testing and therefore may have bad performance in target data analysis. Thus, AutoConsis employs in-context learning (ICL) to enhance LLM's understanding of app testing, which is widely used in LLM tasks [28]. For instance, before actual deployment, testing engineers in Meituan will design multiple example prompts in COT format for each data inconsistency checking task. As will be demonstrated in Section 4, despite the need to design example prompts in COT format, the overall human effort is significantly reduced due to a notable decrease in the rate of LLM's mistakes.

3.4 Consistency Verification

Each inconsistency testing requirement includes the relationships that the target data should satisfy. For example, in the price inconsistency testing, the current price should be equal to the original price minus the discount value. After all the target data is recognized, AutoConsis conducts consistency verification based on these relationships.

Specifically, there are in fact two types of data in commercial mobile apps, *numerical values* and *semantic information in human language*. For numerical values, AutoConsis verifies their consistency via the formula from the given testing requirement. On the other hand, for semantic information, AutoConsis constructs a prompt with the COT mechanism and relies on the understanding and reasoning capabilities of LLMs. Figure 6 shows the prompt for verifying category inconsistency of Figure 1(b). We will provide a detailed elaboration on the use of LLMs through a practical industrial case in Section 5.2.

Example Prompt:

```
There are some textual phrases from a commercial mobile
app:
["category":'Massage', "product name":'Star Shining KTV']

Q: Can you help me determine if these are consistent and
   explain why? Please provide your judgment and reasoning
   in the blanks provided.

- judgement:
- explanation:

A: judgement: Inconsistent.
   explanation:
   According to the product name 'Star Shining KTV', the
   type of product can be 'Karaoke'. Since 'Karaoke' does
   not belong to the category 'Massage', these textual
   phrases from the GUI are inconsistent.
```

+

Target Prompt:

```
There are some textual phrases from a commercial mobile
app:
[$Phrases]

Q: Can you help me determine if these are consistent and
   explain why? Please provide your judgment and reasoning
   in the blanks provided.

- judgement:
- explanation:

A: [To be completed]
```

Figure 6: Prompt for semantic information consistency verification

4 EVALUATION

In this section, we evaluate the performance of AutoConsis by answering the following research questions.

- **RQ1:** How effective is AutoConsis in recognizing diverse GUI sub-regions pages compared with the existing approach?
- **RQ2:** What are the impacts of different parts in the prompt design of AutoConsis?
- **RQ3:** To what extent can AutoConsis detect inconsistency bugs in practical commercial mobile apps?

RQ1 assesses the effectiveness of AutoConsis's multi-modal model, examining its accuracy and efficiency in the sub-region recognition from GUI screenshots. RQ2 leverages the idea of differential testing [2] to evaluate the impact of different parts of AutoConsis's prompt design. In RQ3, we construct a dataset containing 30 inconsistency bugs, simulating practical scenarios to evaluate AutoConsis's usability in data inconsistency bug detection.

4.1 Evaluation Setup

4.1.1 Dataset. To the best of our knowledge, AutoConsis is the first automatic data inconsistency detection tool for commercial mobile apps. Therefore, there isn't an available public dataset suitable for the same purpose. Although vast amounts of data can be

Examples for Standard ICL:

```
Here is a list of phrases from commercial mobile apps:
['Peace Road', 'Garden Cake', 'vanilla cake',
 '66% Sold', 'Dessert', 'Direct subsidy', '45min left',
 '$50', '$30', '$20', '60% off']

Q: Given a text list from commercial mobile apps,
   analyze and extract the following data:

- original price:
- discount:
- price after discount:

A: original price: "50", discount: "30", price after
   discount: "20"
```

Zero-shot Prompt:

```
Here is a list of phrases from commercial mobile apps:
[$Phrases]

Q: Given a text list from commercial mobile apps,
   analyze and extract the following data:

- original price:
- discount:
- price after discount:

A: [To be completed]
```

Figure 7: Prompt baselines

gathered within Meituan, annotating such data demands significant manual effort. Thus, in the evaluation procedure, a small dataset from Meituan's app was employed to validate the design effectiveness of AutoConsis. The use of extensive data in the production environment will be discussed in Section 5.

In industrial practice, AD pages are often related to transactions, the most critical feature for commercial apps. However, due to the frequent updates of commercial apps, these pages typically suffer from low testing coverage. For example, price inconsistencies on AD pages have led to several bugs in Meituan. Given that the data inconsistency bugs in AD pages are not only easy to understand but also under an urgent testing requirement, in this section, we choose AD pages to assess the performance of AutoConsis.

In practice, 130 screenshots of AD pages were collected randomly from three different retail businesses in Meituan, *Dining Voucher*, *Admission Ticket* and *Group Purchase* (*i.e.*, kind of flash sale activities). In Meituan, all of these three businesses have experienced bugs where the current price was inconsistent with the original price minus the discount value. Therefore, the target data for AD pages are the *original price*, *discount value*, and *current price*.

Specifically, 100 screenshots were used to test AutoConsis's performance, while the remaining 30 screenshots will be employed to generate example prompts for in-context learning. For each screenshot, we manually pick out the target sub-regions and then select the target data from them. In order to minimize the influence of human factors during the annotation, two authors of this paper annotate these screenshots independently. If their opinions are not the same, another author will join, and the three can come up with a final decision.

Table 1: Performance comparison of three CLIP-based Models.

Input Type	Precision	Recall	F1-Score
Text	0.266	0.108	0.154
Image	0.714	0.414	0.524
Multi-modal	0.981	1.000	0.991

4.1.2 Baselines. As demonstrated in Section 3.2, traditional object detection methods for image (*i.e.*, YOLOX [12] and YOLOv7 [38]) do not suit diverse GUI pages. Therefore, we leverage a text-only CLIP model and an image-only CLIP model as the baselines for the *GUI Sub-region Recognition* part of AutoConsis. Specifically, the text-only CLIP model takes a textual query (T) to find the GUI sub-regions from a given GUI screenshot. The text-only CLIP model's workflow is the same as AutoConsis except that the image query (V) has been removed. Therefore, its target sub-region $Target_{text}$ is calculated by the following formulas.

$$k = \arg\max_{i=0}^{n}(\cos(E_i, E_t)) \tag{3}$$

$$Target_{text} = s_k \tag{4}$$

Similarly, the image-only CLIP model keeps the image query (V) while removing the textual query (T). Its target-region $Target_{img}$ is represented as:

$$k = \arg\max_{i=0}^{n}(\cos(E_i, E_v)) \tag{5}$$

$$Target_{img} = s_k \tag{6}$$

In order to evaluate the effectiveness of the COT prompt template of AutoConsis, we take zero-shot prompt and standard ICL prompt as baselines. As shown in Figure 7, in the standard ICL prompt, answers in example prompts are directly given. By comparing with the standard ICL, we can evaluate the effectiveness of COT design in AutoConsis. Using the zero-shot prompt, which includes only textual phrases and names of the target data, we aim to investigate the benefits of example prompts in ICL.

4.2 RQ1: Performance of GUI Sub-region Recognition

We conduct an ablation study to evaluate the effectiveness of AutoConsis's multi-modal model in the *GUI Sub-region Recognition* procedure. Specifically, we compare AutoConsis with its two variants (*i.e.*, baselines), the text-only CLIP model and the image-only CLIP model.

Defining the missing target sub-regions as false negative (FN) and the recognized sub-region without target data as false positive (FP), we use Precision (P), Recall (R) and F1-Score (F) to present the performance of the three models.

As shown in Table 1, the multi-modal CLIP model reaches the highest F1-Score among these three models, while the text-only

Table 2: Performance comparison of three prompt templates.

Prompt Type	Dining Voucher				Admission Ticket				Group Purchase				Avg.			
	Acc	P	R	F	Acc	P	R	F	Acc	P	R	F	Acc	P	R	F
Zero-Shot	0.488	0.708	0.717	0.713	0.426	0.640	0.650	0.645	0.608	0.740	0.758	0.749	0.507	0.696	0.708	0.702
Standard ICL	0.712	0.896	0.896	0.896	0.967	0.990	0.990	0.990	0.708	0.894	0.894	0.894	0.796	0.927	0.927	0.927
COT	0.926	0.964	0.964	0.964	0.984	0.995	0.995	0.995	0.907	0.955	0.955	0.955	0.939	0.971	0.971	0.971

CLIP model has the worst performance. Table 1 indicates that both images and text can provide useful information, and can complement each other. Therefore, the multi-modal input design of AutoConsis introduces greater information richness, resulting in enhanced recognition outcomes.

4.3 RQ2: Effectiveness of Prompt Design

In order to demonstrate the effectiveness of the COT prompt template we designed for AutoConsis, we compare AutoConsis with its two variants (*i.e.*, the standard ICL prompt and the zero-shot prompt) with a target data extraction task. Specifically, we evaluate their effectiveness in the extraction of the three price data for each sub-region.

After outputs are generated by ChatGPT [3], we assess the accuracy from both data and product perspectives. Specifically, given that each sub-region includes a product photo and its corresponding textual description, we first calculated how many products had all three prices correctly extracted, denoted as *Acc*. Additionally, from the perspective of target data, we also leverage the Precision (P), Recall (R) and F1-Score (F) to represent the accuracy of the price data extraction.

Table 2 shows the performance of different prompts. The outputs of the zero-shot prompt exhibit the poorest performance. This is caused by the zero-shot prompt leading the LLM to generate many incorrect "target data", missing some real target data in the meantime. The standard ICL outperforms zero-shot, suggesting that example cases in prompt can assist the LLM in understanding the current task. With the incorporation of the COT mechanism, AutoConsis has the best performance, suggesting that COT examples can effectively guide the LLM in handling complicated logical tasks.

4.4 RQ3: Inconsistency Bugs Detection

4.4.1 Inconsistency Detect Experiment. Based on the historical bugs in Meituan, by modifying GUI screenshots, two authors of this paper manually created 30 data inconsistency bugs (10 bugs for each business) by changing any one or two among the original price, discount, or current price of 30 products. To assess the false positive rate of AutoConsis for correct cases, we randomly selected 40 products with consistent current prices for each business. Therefore, the dataset for RQ3 comprises 20% of bug cases.

Subsequently, we tested whether AutoConsis could fully detect these 30 bugs and checked for any false positives.

Table 3 shows the detection result. AutoConsis successfully detected 28 buggy pages among the 30 bug cases achieving a recall of 93.3%. The main reason for these unrecognized cases is that the LLM overly relies on its inference capabilities, and occasionally modifies incorrect prices during the target data extraction process.

Table 3: Data Inconsistency Bugs Detection Performance

Business	#Bugs	#Rec.	Recall	#FPs	FPr
Dining Voucher	10	9	0.900	1	0.025
Admission Ticket	10	10	1.000	2	0.050
Group Purchase	10	9	0.900	2	0.050
Total	30	28	0.933	5	0.042

[1] "#Rec." represents the number of the bugs recognized by AutoConsis.

4.4.2 False-Positive Analysis. As shown in Table 3, there are 5 false positive (FP) cases in AutoConsis's output.

Given that the consistency certification of the three price numbers can be conducted simply by a mathematical formula, all of these FP cases are caused by mistakes in the target data extraction procedure. Specifically, since the false sub-region created by the multi-modal CLIP model could be parsed with their irregular position, all of these FP cases are caused by LLM's mistakes.

In industry practice, the proportion of FPs (*i.e.*, the FPr), that require manual verification, is crucial. Before the deployment of AutoConsis, we invited testing engineers from Meituan to assess its feasibility. Given the low FPr in AutoConsis's results and identifying a false positive takes at most three times longer than a regular case, we estimate that AutoConsis can reduce manual efforts in data inconsistency testing by 87%.

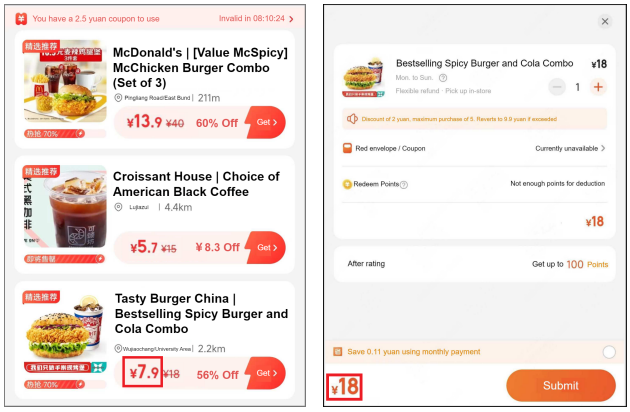
5 CASE STUDIES AND DISCUSSIONS

Each part of AutoConsis was developed within Meituan and has now been integrated into an automated testing platform for testing engineers' easy access. When invoking AutoConsis, engineers are required to provide a list of pages to be tested, examples of several sub-regions for automatic UI pre-processing, and to complete the missing parts of the COT Prompt Template for target data analysis. With such information, AutoConsis can then autonomously conduct data inconsistency detection across all the pages.

In this section, we present cases to introduce the practical benefits of AutoConsis on commercial mobile app testing in Meituan.

5.1 Case 1: False Advertising Detection

Serving over 600 million users, Meituan places a high priority on user experience within its apps. In particular, to ensure the stability of its core business, most of the transaction-related pages in Meituan's mobile app require multiple rounds of testing (*i.e.*, regression testing, script testing and manual testing). Moreover,



(a) AD page with appealing current price (b) Order-placement page with no discount

Figure 8: AD-order Price Inconsistency Case

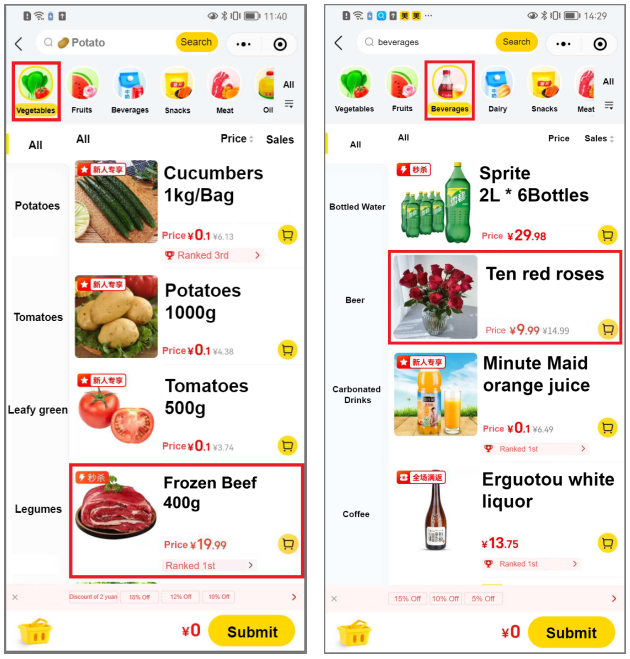
since the price of products is a sensitive factor for customers, significantly influencing their purchasing decisions and even their choice of shopping platforms, price-related bugs in Meituan are typically treated with urgency.

Take the customer's shopping process in Meituan's app as an example. Customers are typically first attracted by the products and prices on the **advertisement page** such as Figure 8(a). By clicking on the advertisement card, they enter the product detail page shown in Figure 8(b) to view further information about the product. Then, on the **order placement page**, customers will choose the quantity of products and confirm the final price. Typically, the final price is calculated by subtracting the discount from the original price of the product. If there is a price inconsistency between the **advertisement page** and the **order placement page** (*i.e.*, false advertising), it will severely impact the user experience, tarnish the company's reputation, and lead to customer attrition.

Although addressing AD-order price inconsistencies is urgent, Meituan's strategy of deploying AD pages tailored to local shops and attractions across hundreds of cities and districts results in a vast array of unique AD pages. This diversity has historically made widespread detection of AD-order price inconsistencies a challenging task.

Therefore, AutoConsis was first deployed to conduct the automatic detection of AD-order price inconsistencies. Specifically, 5,000 AD pages were collected from more than 754 different cities or districts. For each AD page, AutoConsis first recognized all of the AD cards (*i.e.*, AD for one product) and found out the current price $Price_{AD}$ on each AD card. Then, by clicking the "Get" button via the Android Debug Bridge (adb) tool [1], AutoConsis navigated to the order placement page and extracted the final price $Price_{order}$ in it. Finally, AutoConsis conducted the AD-order price inconsistency analysis by checking whether $Price_{AD}$ equals $Price_{order}$.

AutoConsis found 10 distinct AD-order price inconsistency bugs among the 5,000 AD pages, and 8 were confirmed by testing engineers in Meituan. To prevent these bugs from appearing again, we



(a) Beef is on the vegetable shelf (b) Flowers categorized under "Beverages"

Figure 9: Product-shelf Mismatch Cases

traced the causes of the 8 bugs: two were due to API response timeouts, one was caused by incorrect parameter transmission (missing parameters), and the remaining five bugs were caused by the shortage of discount APIs invoking on advertisement pages. Currently, all of these eight bugs have been resolved.

5.2 Case 2: Verifying Product-Category Consistency

Given that numbers are just the simplest form of data, discrepancies in the natural language concepts can lead to more complex data inconsistency bugs. Compared with numeric data inconsistencies, such bugs are more difficult to detect. In the past, it could only be detected through manual inspection.

In the past, Meituan experienced a serious online problem: in the "Grocery" retail business, many products were mistakenly placed on the wrong shelves. As shown in Figure 9(a), beef is put on the "Vegetable" shelf. This weird occurrence confused customers and decreased the regular sales of the Grocery business that day. Upon analysis, we found that in the Grocery business, products not only have seasonality updates, but also frequently vary due to promotional activities from suppliers. In each update, the categories of most products are determined manually, which consistently results in product-category mismatches within the Grocery business.

To prevent such product-category mismatches from arising again, we deployed AutoConsis to the grocery list pages of Meituan's mobile apps and periodically conducted product-category inconsistency detection. Specifically, since products ranked higher tend to be viewed more frequently by customers, we collected GUI pages

from Meituan's app that featured the first 100 products from each shelf within the Grocery business. AutoConsis first identifies GUI sub-regions containing each product's information (*i.e.*, a product photo, name and corresponding textual introduction). It then constructs a COT prompt with the template in Figure 5, utilizing LLM to extract the product's name from each sub-region.

As for the consistency verification procedure, determining the consistency between product names and categories requires inferential reasoning. Hence, AutoConsis constructs a COT prompt with the template in Figure 6 and leverages LLM to accomplish this check.

5.3 Threats to Validity

One possible threat to validity is whether the AutoConsis design is specifically tailored for our target apps (that in Meituan) and cannot work well for other popular commercial apps. Although the design and validation of AutoConsis were conducted within Meituan, functional bugs characterized by data inconsistency are prevalent across various commercial mobile apps. This formulation is also one of the core contributions of this paper. Additionally, by mimicking human behavior, AutoConsis addresses the challenges caused by GUI diversity, which is a common characteristic of commercial apps [39, 40]. Hence, the design of AutoConsis can shed light on the automatic functional testing of other mobile apps.

6 RELATED WORK

Performing automated testing of mobile apps attracted academic interest for a long time. In the literature, many GUI testing approaches have been proposed based on various techniques [35, 40], including random testing [7, 16, 24], evolutionary algorithms [25, 27], model-based testing [13, 34], and systematic exploration [9, 10]. Most proposed approaches use app exceptions as implicit test oracles, and only focus on detecting non-functional faults, *e.g.*, crashes and not-responding errors [14, 20, 33]. Nevertheless, bugs in mobile apps may not necessarily manifest themselves as non-functional faults.

In order to effectively test mobile apps with complicated, diverse business logic, current industrial practices heavily rely on manual validation, which incurs substantial costs. Due to the labor-intensive nature of current app testing, both academics and industry spent much effort on finding functional bugs. AppFlow [17] and CraftDroid [21] leverage human-provided testing directions to find non-crashing bugs. Different from these works, AutoConsis does not require such specific testing directions, resulting in less manual intervention. There are also some tools that focus on a specific class of functional bugs. DiffDroid [11] can detect device-or platform-specific bugs by comparing the GUI images or screenshots (explored by the same GUI test). SetDroid [36] can generate automated test oracles for system setting-related, non-crashing defects of Android apps, and ACETON [18] focus on energy bugs' automatic detection. Unlike these approaches, AutoConsis focuses on the data which is integral to all functionalities within a mobile app. By mimicking the process of human recognition, AutoConsis is capable of detecting various data inconsistency bugs.

Given their vast knowledge base and logical reasoning capabilities, large language models (LLMs) have also been widely adopted for mobile app testing. DroidBot-GPT [44] and AutoDroid [43] leverage LLMs generated UI action to conduct Android UI automation. QTypist [22] concentrates on the generation of semantic textual input for form pages of mobile apps. Different from these tools, AutoConsis can directly detect functional bugs in complicated mobile apps (*e.g.*, popular commercial apps). Moreover, through case studies on thousands of practical GUI pages, we demonstrate that AutoConsis can effectively reduce manual labor costs in functional bug detection of industrial practice.

7 CONCLUSION

In this paper, we propose AutoConsis, an automatic approach to detect data inconsistency bugs for mobile apps. In order to extract target data from GUI pages, AutoConsis adopts a special-tailored CLIP model to recognize GUI sub-regions containing target data. Then, AutoConsis constructs a COT prompt and extracts target data with the help of LLM. Finally, AutoConsis detects inconsistency bugs by verifying relationships between these target data. Our experiments show that AutoConsis outperforms other methods in data extraction from GUI pages. Furthermore, extensive case studies show the practical benefits of using AutoConsis to detect data inconsistency bugs for complex commercial apps.

ACKNOWLEDGMENTS

This work was supported by Meituan and the Shanghai Municipal Science and Technology Commission (Project No. 22DZ1204900 and No. 22ZR1407900). In addition, we extend our heartfelt thanks to our colleagues in Meituan, specifically, Jia, Jie, Chen, Huabin and Xianxuan, for their kind help and support in this work. Y. Zhou is the corresponding author.

REFERENCES

[1] Accessed: 2023. *Android Debug Bridge (adb).* https://developer.android.com/tools/adb
[2] Accessed: 2023. *Differential testing.* https://en.wikipedia.org/wiki/Differential_testing
[3] Accessed: 2023. *Introducing ChatGPT.* https://openai.com/blog/chatgpt
[4] Accessed: 2023. *Layouts in Views.* https://developer.android.com/develop/ui/views/layout/declaring-layout
[5] Accessed: 2023. *Meituan.* https://about.meituan.com
[6] Accessed: 2023. *Meituan ANNUAL REPORT 2022.* http://media-meituan.todayir.com/20230425215252173578660 4_en.pdf
[7] Accessed: 2023. *UI/Application Exerciser Monkey.* https://developer.android.com/studio/test/other-testing-tools/monkey
[8] Accessed: 2023. *XML Path Language (XPath).* https://www.w3.org/TR/1999/REC-xpath-19991116/
[9] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proc. of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE.* ACM, 59.
[10] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proc. of the 42nd International Conference on Software Engineering, ICSE.* ACM, 481–492.
[11] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE.* IEEE Computer Society, 308–318.
[12] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. 2021. YOLOX: Exceeding YOLO Series in 2021. *CoRR* abs/2107.08430 (2021).
[13] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proc. of the 41st International Conference on Software Engineering, ICSE.* IEEE / ACM, 269–280.
[14] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. 2020. Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis.

In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 557–568.

[15] Alain Horé and Djemel Ziou. 2010. Image Quality Metrics: PSNR vs. SSIM. In *20th International Conference on Pattern Recognition, ICPR 2010, Istanbul, Turkey, 23-26 August 2010*. IEEE Computer Society, 2366–2369.

[16] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proc. of the 6th International Workshop on Automation of Software Test, AST*. ACM, 77–83.

[17] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proc. of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 269–282.

[18] Reyhaneh Jabbarvand, Forough Mehralian, and Sam Malek. 2020. Automated construction of energy test oracles for Android. In *Proc. of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering,ESEC/FSE*. ACM, 927–938.

[19] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated test generation for REST APIs: no time to rest yet. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 289–301.

[20] Pingfan Kong, Li Li, Jun Gao, Timothée Riom, Yanjie Zhao, Tegawendé F. Bissyandé, and Jacques Klein. 2021. ANCHOR: locating android framework-specific crashing faults. *Autom. Softw. Eng.* 28, 2 (2021), 10.

[21] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 42–53.

[22] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1355–1367.

[23] Michael R Lyu et al. 1996. *Handbook of software reliability engineering*. Vol. 222. IEEE computer society press Los Alamitos.

[24] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Proc. of 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE*. ACM, 224–234.

[25] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proc. of the 22nd ACM SIGSOFT Symposium on Foundations of Software Engineering, FSE*. ACM, 599–609.

[26] Tarek Mahmud. 2021. API Compatibility Issue Detection, Testing and Analysis for Android Apps. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1061–1063.

[27] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 94–105.

[28] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 11048–11064.

[29] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael R. Lyu. 2023. Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors. *CoRR* abs/2306.01394 (2023).

[30] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, Marina Meila and Tong Zhang (Eds.). PMLR, 8748–8763.

[31] Dezhi Ran, Zongyang Li, Chenxu Liu, Wenyu Wang, Weizhi Meng, Xionglin Wu, Hui Jin, Jing Cui, Xing Tang, and Tao Xie. 2022. Automated Visual Testing for Mobile Apps in an Industrial Setting. In *Proc. of the 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*. IEEE, 55–64.

[32] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580* (2023).

[33] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2022. Why My App Crashes? Understanding and Benchmarking Framework-Specific Exceptions of Android Apps. *IEEE Trans. Software Eng.* 48, 4 (2022), 1115–1137.

[34] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 245–256.

[35] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 119–130.

[36] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *Proc. of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*. ACM, 204–215.

[37] Vidit Vidit, Martin Engilberge, and Mathieu Salzmann. 2023. CLIP the Gap: A Single Domain Generalization Approach for Object Detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*. IEEE, 3219–3229.

[38] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2023. YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art for Real-Time Object Detectors. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*. IEEE, 7464–7475.

[39] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An infrastructure approach to improving effectiveness of Android UI testing tools. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 165–176.

[40] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 738–748.

[41] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. Vet: identifying and avoiding UI exploration tarpits. In *Proc. of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering,ESEC/FSE*. ACM, 83–94.

[42] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.

[43] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2023. Empowering LLM to use Smartphone for Intelligent Task Automation. *CoRR* abs/2308.15272 (2023).

[44] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. DroidBot-GPT: GPT-powered UI Automation for Android. *CoRR* abs/2304.07061 (2023).