

# OmniACT: A Dataset and Benchmark for Enabling Multimodal Generalist Autonomous Agents for Desktop and Web

Raghav Kapoor<sup>†♦</sup> Yash Parag Butala<sup>†♦</sup>  
 Melisa Russak<sup>♣</sup> Jing Yu Koh<sup>♦</sup> Kiran Kamble<sup>♣</sup>  
 Waseem AlShikh<sup>♣</sup> Ruslan Salakhutdinov<sup>♦</sup>  
<sup>♦</sup>Carnegie Mellon University <sup>♣</sup>Writer.com  
 {raghavka, ypb}@cs.cmu.edu



Figure 1. **OmniACT** dataset and benchmark for enabling autonomous human-computer interaction agents. The left shows an image paired with a natural language task description as the input, the right shows the resulting action script to be executed on the screen. Examples are presented from Stocks, Apartments.com, and Weather application.

## Abstract

For decades, human-computer interaction has fundamentally been manual. Even today, almost all productive work done on the computer necessitates human input at every step. Autonomous virtual agents represent an exciting step in automating many of these menial tasks. Virtual agents would empower users with limited technical proficiency to harness the full possibilities of computer systems. They could also enable the efficient streamlining of numerous computer tasks, ranging from calendar management to

<sup>†</sup> These authors contributed equally. The order is determined by dice rolling.

complex travel bookings, with minimal human intervention. In this paper, we introduce **OmniACT**, the first-of-a-kind dataset and benchmark for assessing an agent’s capability to generate executable programs to accomplish computer tasks. Our scope extends beyond traditional web automation, covering a diverse range of desktop applications. The dataset consists of fundamental tasks such as “Play the next song”, as well as longer horizon tasks such as “Send an email to John Doe mentioning the time and place to meet”. Specifically, given a pair of screen image and a visually-grounded natural language task, the goal is to generate a script capable of fully executing the task. We run several strong baseline language model agents on our bench-

*mark. The strongest baseline, GPT-4, performs the best on our benchmark However, its performance level still reaches only 15% of the human proficiency in generating executable scripts capable of completing the task, demonstrating the challenge of our task for conventional web agents. Our benchmark provides a platform to measure and evaluate the progress of language model agents in automating computer tasks and motivates future work towards building multimodal models that bridge large language models and the visual grounding of computer screens.*

## 1. Introduction

Performing computer tasks based on natural language instructions has been a long-standing goal of artificial intelligence [45]. One concrete objective in the line of research is to develop generalist agents that can assist humans in doing computer tasks [19], such as “*Order a pizza from Domino’s*” or “*Write a message to John.*” The agent should be able to open the application and perform the task. Executing these actions on a personal computer involves a sequence of interactions with a mouse and keyboard. For example, the simple task of writing an email involves hovering over the application icon, clicking it, clicking the ‘*New Email*’ button, writing the content of the email, and clicking send. Successfully sending an email requires accurately predicting the correct action at each step and accurately executing it, which is a herculean task even for the best agents today [13].

A generalist agent for computer tasks must understand natural language instructions, process visual screenshots, and produce the correct sequence of actions to be performed to achieve the intended task. Several existing approaches focus on building agents based on the HTML model [9, 37, 57]. However, this approach introduces several challenges and constraints. These agents are limited to web applications and often struggle with complex or long-context HTML code. They cannot interact with native desktop applications or perform tasks that span multiple applications, like drafting an email using text from a code editor, without significant alterations. Furthermore, HTML-based agents, which are inherently powered by text-only language models, typically underperform in tasks requiring visual cues, such as identifying and clicking a blue button on a desktop’s top-right corner. In contrast, humans can easily understand UI elements like dropdown menus, typable areas, redirections, and options with just a glance.

Towards the goal of developing a generalist autonomous agent with robust visual and user interface (UI) understanding capabilities, we introduce a new task and dataset, **OmniACT**, containing over 9.8K pairs of images and instructions (Figure 1) across different operating systems and the web. This dataset includes screenshots of various UI screens and corresponding natural language instruc-

tions. The objective of these instructions is to generate executable commands using the *PyAutoGUI* Python library [1]. *PyAutoGUI* enables the automation of the mouse and keyboard operations, which helps to facilitate interactions with various native applications across macOS, Windows, and Linux. This simplifies completing specified tasks across different web domains and native desktop applications.

We evaluate several language model-based agent baselines on this dataset, including LLaMA [43], Vicuna [7], Palmyra-X (43B) [2], InstructPalmyra-30B [41], GPT 3.5, and GPT-4 [30]. We experiment with fine-tuning Vicuna-13B and LLaMA-13B models using QLoRA [10]. We also benchmark multimodal baseline LLaVa-v1.5-7B, LLaVa-v1.5-13B [43] and GPT-4-vision-preview [50] for the task. Our findings highlight the necessity for a multimodal model capable of executing these tasks, and our analysis provides insights into promising future work in the space. Our key contributions are outlined as follows:

1. We release a novel dataset of desktop and website applications consisting of over 9.8K natural language tasks, UI screens, and corresponding code snippets collected through human annotation. We introduce custom performance metrics tailored for computer tasks.
2. We propose DetACT, a module for creating textual representations of the screen using signals from OCR, color, and icon-template matching.
3. We conduct a comprehensive benchmark and analysis of state-of-the-art LLMs and multimodal models on our benchmark. Our results show that **OmniACT** is a challenging task for even the best LLM agents today, and existing models are far below human performance.

## 2. Related Work

### 2.1. UI Understanding

User interface (UI) understanding has garnered interest from researchers in the machine learning and human-computer interaction communities, evolving with various models focusing on understanding the semantics of mobile and web user interfaces. UIBert [3], PixelBERT [15], ActionBert [14], VUT [23], Screen2Words [44], Widget-Captioning [22] and Pix2Act [36] are notable models in this area. They propose approaches for learning the user-interface semantics of the mobile screen using the image and view hierarchy. These models have demonstrated effectiveness in tasks like capability prediction, screen segmentation and understanding, and screen caption generation. Lexi [4] and Spotlight [20] propose models that use vision-only inputs to minimize the reliance on metadata such as view hierarchy. Furata et al. [11] demonstrates the use of fine-tuning for multimodal web navigation. The majority of machine learning models trained for UI understanding leverage the Rico dataset [8] and its extensions, which contain 64,462 unique Android screens and meta-

Datasets	Size	Env Type	Task Heterogeneity	Real-World Portayal	Executional Correctness	Supports Desktop Apps	Continuous Scale Adaptive Evaluation	Task
WebArena [57]	812	Web	Yes	Yes	Yes	No	No	Web Navigation
Mind2Web [9]	2350	Web	Yes	Yes	No	No	No	Web Navigation
WebShop [51]	12000 Products	Web	No	No	Yes	No	No	Web Navigation
RUSS [49]	80	Web	Yes	Yes	No	No	No	Web Navigation
WebSRC [6]	2735	Web	Yes	Yes	-	No	No	QA
MiniWoB++ [16]	100	Mobile Websites	No	No	Yes	No	No	Web Navigation
PixelHelp [21]	187	Mobile	Yes	Yes	No	No	No	UI Grounding
MetaGUI [39]	1125	Mobile	Yes	Yes	Yes	No	No	Mobile Navigation
MoTIF [5]	756	Mobile	Yes	Yes	Yes	No	No	Mobile Navigation
AITW [32]	715142	Mobile and Web	Yes	Yes	Yes	No	No	Mobile/Web Navigation
<b>OmniACT (Ours)</b>	<b>9802</b>	<b>Desktop and Web</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Code Generation</b>

Table 1. Comparison of OmniACT with other related benchmarks.

data. In addition, Banerjee et al. [4] released the UICaptions dataset, which consists of diverse image-captions pairs across a wide range of applications. PixelHelp [21] also released a corpus to train models that can interpret natural language instructions and map them to mobile user interface actions.

## 2.2. Autonomous Computer Agents

The advent of large language models (LLMs) has been pivotal in the rapid advancement of agents that operate on web pages. Recent research such as ViperGPT [40] Chameleon [26], RCI Agent [17], VisProg [12], and [28] employ LLMs for planning or action prediction in developing autonomous agents. Benchmark datasets, such as MiniWoB [37], WebShop [51], Macaw-LLM [27], ASH-Prompting [38] Mind2Web [9] and WebArena [57] have also been proposed to measure the ability of LLM-based agents in automating web tasks. These methods mainly involve agents that operate on a text-based Document Object Model (DOM) of HTML scripts. This limits their understanding of screen context, which is crucial for the model’s decision-making and action-taking processes. To address this limitation, Rawles et al. [32] released Android in the Wild, a dataset comprising screens, natural language instructions, and corresponding actions. Following this, [54] proposed a multimodal model, AutoUI, which is designed to build an agent on the Android in the Wild dataset confined to the Android ecosystem.

Current benchmarks for autonomous agents focus mainly on the Web or Android environments, posing challenges for tasks involving desktop applications or spanning multiple applications beyond the web domain. The absence of established benchmarks and datasets in this area, coupled with basic methods for extracting user interface (UI) elements, underscores the need for significant progress in developing more versatile autonomous agents capable of handling diverse tasks beyond the current scope. To highlight the unique features that **OmniACT** introduces in the assessment of capable autonomous agents, we provide a com-

parison between the existing benchmarks and our proposed benchmark, **OmniACT**, in Table 1.

## 3. OmniACT

We introduce a novel dataset and benchmark, **OmniACT**, which measures the performance of autonomous agents on both web and desktop applications. Compared to previous benchmarks which focus on text-based reasoning [9, 16, 37, 51, 57], our benchmark aims to measure multimodal agents that bridge large language model planners and UI understanding vision models. **OmniACT** can be accomplished as a standalone task as it is not under a mock environment.

All actions that a human can execute on the computer can be encoded in the *PyAutoGUI* [1] Python framework. This framework allows a user to execute keyboard and mouse operations by running Python code. The *PyAutoGUI* code to execute these tasks is shown in the third column of Figure 1. For other computer tasks, the *PyAutoGUI* library provides functions such as ‘press’, ‘write’, and ‘scroll’ which can be used to execute the task. Our dataset consists of parallel data of natural language tasks, UI screenshots, and ground truth *PyAutoGUI* scripts that achieve successful execution.

### 3.1. Task Formulation

Given an input state of a computer defined by the screen  $S$  and the task description  $T$  in natural language, the goal of the task is to output a sequence of actions  $A$  that can successfully accomplish the task  $T$  within a screenshot  $S \in \{\text{Linux, Windows, MacOS, Webpage}\}$ . Formally, the task can be defined as learning the transition function  $f : T \times S \rightarrow A$ . During dataset collection, we ensure that all task descriptions  $T$  are feasible and can be accomplished in the current screenshot  $S$ . To reduce ambiguity and facilitate better evaluation, we ensure that task descriptions are detailed and unambiguous. Tasks can also be visually grounded (e.g., ‘Click the red button to start recording’) or natural language based (e.g., ‘Click the My Account button’). We define the action space using the functionalities in the *PyAutoGUI* library:

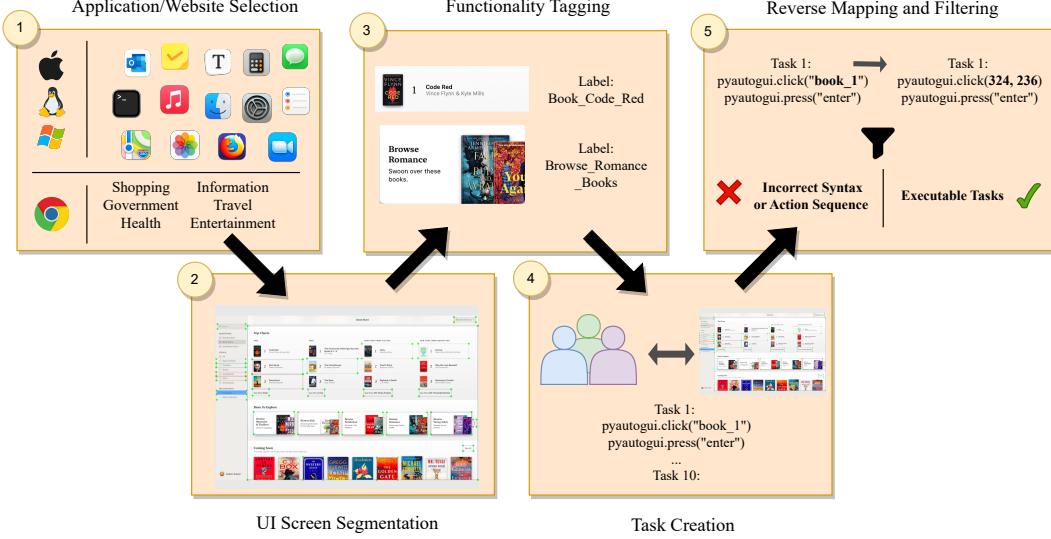


Figure 2. **Data Collection Pipeline.** (1) We select over 60 applications and websites to ensure diversity, (2) segment the screen through human-annotated bounding boxes, (3) label the bounding boxes based on functionality, (4) ask student volunteers to come up with tasks, given a screen image, and (5) reverse map the textual labels to coordinates and filter the scripts based on execution and syntax.

$A \in \{\text{'click'}, \text{'dragTo'}, \text{'scroll'}, \text{'write'}, \dots\}$ . The exhaustive list of actions is provided in Table 2. Our action space is much larger than other benchmarks [9, 37, 57] that resort to two or three interaction options. Mouse actions such as ‘moveTo’, ‘click’, ‘rightClick’, ‘doubleClick’, and ‘dragTo’, additionally require screen coordinates as arguments, which indicate the pixel location of the action.

Figure 1 illustrates sample tasks and corresponding outputs for three applications within OmniACT: (1) Stocks (MacOS), (2) Apartments.com (web page), and (3) Weather (MacOS). The first column depicts the input image, and the second column shows the natural language task that is to be executed on the current screen. To execute these tasks, a user must accurately perform a series of operations using the mouse and keyboard. For example, to check the rate of change in Google’s stock price over the last month, the mouse has to be moved to the last month and dragged while holding the left-click button to the current month.

### 3.2. Dataset Preparation

To prepare our dataset, we followed a pipelined approach, as summarized in Figure 2. We first selected a variety of applications and websites. For each application or website, we created bounding boxes around key UI elements and labeled them according to their functionality, which is crucial for assisting human annotators in writing accurate *PyAutoGUI* scripts. After each script is written, we converted the labels back into numeric coordinates, allowing us to align the scripts precisely with the locations of the UI elements. Finally, we thoroughly reviewed each script, focusing on its executability and adherence to syntax standards. This ensured the high quality and functionality of our dataset,

Type	Action	%
Mouse	Click	63.73
	Double Click	0.58
	Right Click	0.77
	Move/Hover	1.85
	Drag	0.29
	Scroll	1.68
	Horizontal Scroll	0.17
Keyboard	Press	16.28
	Hotkey	3.00
	Write	11.65

Table 2. Action type supported by OmniACT and the number of instances for each of the actions in the dataset.

making it a valuable resource for training and evaluating autonomous agents.

#### 3.2.1 Application/Website Selection

To test the computer agents’ generalization ability across different tasks, we collect tasks across multiple domains on both desktop and web applications. In total, we collect and annotate 9802 data points (Table 3), with the split between desktop and web applications approximately 3:1. The emphasis on desktop applications, which do not contain Document Object Model (DOM) hierarchies unlike HTML-based web pages, presents a more complex multimodal challenge where visual cues are crucial. We collect tasks from applications within the three most popular operating systems. We select 22 native applications from MacOS, and 8 each from Linux and Windows. We annotate roughly 3 to 4 screens for every application. The full list of applications is provided in the Appendix.

Many common computer tasks today are still performed

through web applications, so we also collect 3-4 screenshots from 27 different web applications. To ensure diversity in task intents, we categorize these tasks into one of the following 6 categories: (1) Shopping, (2) Entertainment, (3) Service, (4) Government, (5) Travel, (6) Health. Inspired by the methodology of [9], these categories were selected to cover a wide range of user intents and functionalities.

### 3.2.2 UI Screen Segmentation

To collect gold-standard data, we first annotate and segment the screen by identifying the bounding boxes present on the screen. We employ slightly different techniques for web and desktop applications to create the bounding boxes:

- Desktop Applications:** We build a custom annotation interface based on PyQt5<sup>1</sup> to create bounding boxes manually over a screen image using a simple drag-and-click mechanism. This custom interface expedites the process and allows us to get highly accurate gold-label data points for desktop images.
- Websites:** For webpages, we write JavaScript code to extract all interactable (click, hover, type, etc.) regions from the HTML source code. We also extract banners, dropdowns, submit, and radio buttons from the screen. We filter the elements to retain only those that are visible and interactable within the screen.

### 3.2.3 Functionality Tagging

To map each bounding box to its correct functional description, we leverage Amazon MTurk workers (see details in Appendix), who are given an image with a bounding box and are required to write the correct description or label of the bounding box’s function. For example, given an image of an Amazon webpage with a *search bar*, the annotator labels it as “*find-product-search-bar*”. The logical descriptions are used to create tasks in a structured manner without the need to identify individual bounding box coordinates.

### 3.2.4 Task Creation

Our approach for each screen involves utilizing all human-annotated bounding boxes and their labels to create tasks that can be executed within the confines of a single screen. These tasks are designed to be visually grounded in order to measure the capabilities of multimodal agents. We plan to release the bounding box and their corresponding labels as the metadata for evaluation purposes.

For dataset compilation, college students with basic Python programming skills served as annotators, accessing API references for *PyAutoGUI* and examples of potential tasks. Each student generated multiple tasks, each accompanied by three alternative natural language reformulations. For instance, “*What is 3+2?*” might be reformulated as

<sup>1</sup><https://pypi.org/project/PyQt5/>

Domain		Train	Validation	Test	Total
Desktop	Mac OS	3028	444	786	4258
	Linux	761	126	247	1134
	Windows	1573	216	458	2247
Web	-	1427	206	530	2163
<b>Total</b>		6789	992	2,021	9802

Table 3. Dataset distribution across splits and platforms.

“Calculate the sum of 2 and 3” or “Add two to three”. To avoid train-test leakage, rephrased tasks were consistently placed in the same dataset split. Further details on the annotation process are available in the Appendix.

### 3.2.5 Reverse Mapping and Filtering

To ensure high-quality data, we incorporate an additional step into the data collection pipeline. We build scripts to map the text-based labels of each bounding box back to their numeric coordinates, and then match the syntax and verify if the task will be executed on the screen. Using this filter, we remove all the non-working or syntactically incorrect data points and finally manually review the set of tasks.

After filtering, we obtain 9802 human-annotated, gold-label data points across more than 200 desktop and web screens (Table 3), split into train, validation, and test sets in a 7:1:2 ratio. All collected data will be publicly released to encourage future work on multimodal agents.

## 4. Evaluation Metrics

In this section, we detail various evaluation metrics for benchmarking model performance on the *OmniACT* dataset. UI screens have additional constraints such as spatial relevance which are not factored in most conventional similarity-based metrics such as BLEU [31], Code-BLEU [33], BERTScore [53] and CodeBERTScore [56]. For example, a valid click action is usually not constrained to a single coordinate but can be any coordinate within a specified region. In the event of invalid coordinate predictions, an agent that predicts coordinates further away from the valid region should invoke a higher penalty compared to an agent that predicted coordinates close to the region. We propose two new metrics adapted: Sequence Score (Section 4.1) and Action Score (Section 4.2) aimed at utilizing UI information.

### 4.1. Sequence Score

The sequence score measures whether the predicted action sequence (e.g., ‘click’, ‘write’, ‘press’) exactly matches the gold sequence. Since predicting the first action in the sequence is relatively straightforward and later actions are more difficult, we define sequence score as follows:

$$SeqScore_i = \begin{cases} \beta_1 + \beta_2 * (s - 1) & \text{if all actions match} \\ 0 & \text{otherwise} \end{cases}$$

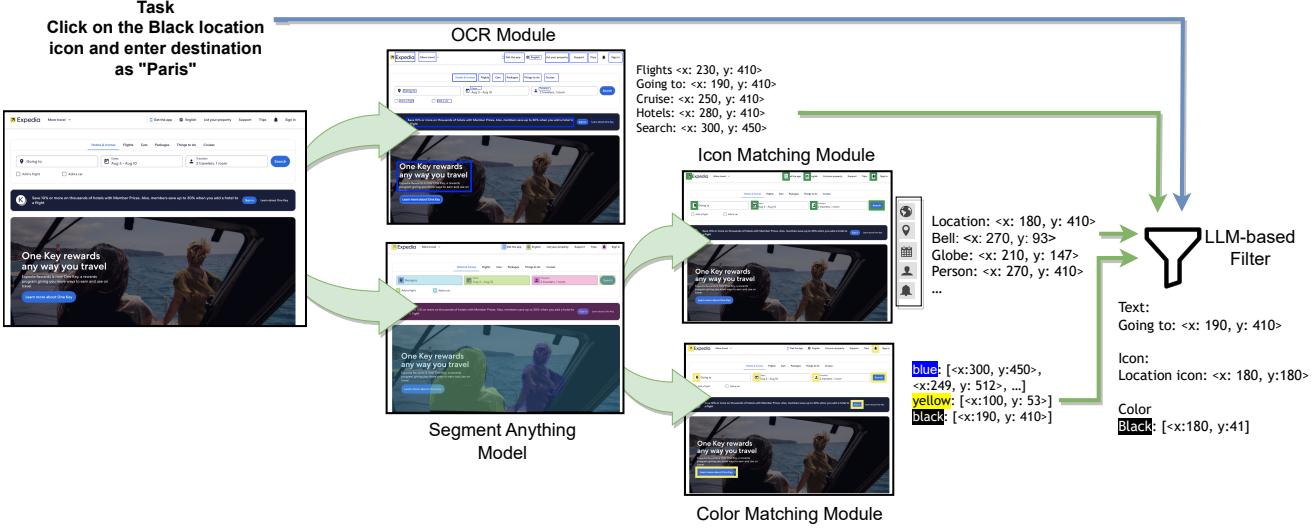


Figure 3. **DetACT Module.** Given an initial image and a natural language task description, we use a pipelined approach to run OCR and SAM on the screen. The outputs from SAM are then used by icon and color-matching modules to obtain an exhaustive set of useful UI elements. The list of elements is passed through LLM based filter to select only the elements related to the given task.

where  $s$  is the action sequence length,  $\beta_1$  is set to 0.1 and  $\beta_2$  is set to 1.

## 4.2. Action Score

The action score measures how well a code snippet containing the correct action sequence can perform the task. Specifically, for a script with a correct action sequence, we introduce penalties for inaccurate behavior. The penalties are described below:

1. **Click penalty ( $M$ ):** For actions ‘click’, ‘rightClick’, ‘doubleClick’, ‘moveTo’, and ‘dragTo’, we penalize code snippets where predicted coordinates lie outside of the bounding box of the UI element. The click penalty for the  $j^{th}$  action of the  $i^{th}$  example is defined as:

$$M_i^j = \alpha_i \times \begin{cases} 1 - \frac{\mu}{\mu + L_2} & \text{if } SeqScore_i > 0 \\ 1 & \text{otherwise} \end{cases}$$

Here  $L_2$  corresponds to the smallest Euclidean distance between the predicted coordinate and bounding box.  $L_2$  is zero when the predicted coordinate lies within the target bounding box.  $\mu$  is the Dirichlet smoothing coefficient which we dynamically set to the inverse of the length of the diagonal of the bounding box. This ensures that the penalty for points outside the bounding box varies based on the size of the bounding box. For two predicted points with the same  $L_2$ , the metric penalizes more heavily if the box is larger. This is sound with the intuition that the chances of clicking on a larger box are higher and should be penalized more in case of a mistake.

2. **Key penalty ( $K$ ):** For actions ‘press’ and ‘hotkey’, we check whether the set of keys in the target code (rep-

resented as  $GK_i^j$ ) and predicted code (represented as  $PK_i^j$ ) are the same. It is formally defined as:

$$K_i^j = \alpha_i \times \begin{cases} 0 & \text{if } GK_i^j = PK_i^j \text{ and } SeqScore_i > 0 \\ 1 & \text{otherwise} \end{cases}$$

3. **Write penalty ( $W_p$ ):** For action type ‘write’, we penalize the output for the sentence to be typed. Specifically, we employ BLEU score [31], and compute:

$$W_i^j = \alpha_i \times \begin{cases} 1 - BLEU(GS_i^j, PS_i^j) & \text{if } SeqScore_i > 0 \\ 1 & \text{otherwise} \end{cases}$$

Here,  $GS_i^j$  represents the actual sentence to be typed, and  $PS_i^j$  represents the sentence predicted by the model in the  $j^{th}$  action of example  $i$ .

In the above equations,  $(\alpha_i)$  is the weighting factor:

$$\alpha_i = SeqScore_i / \text{length of sequence } i$$

This ensures that the action score  $\in [0, 1]$ . The mean action score is calculated as follows:

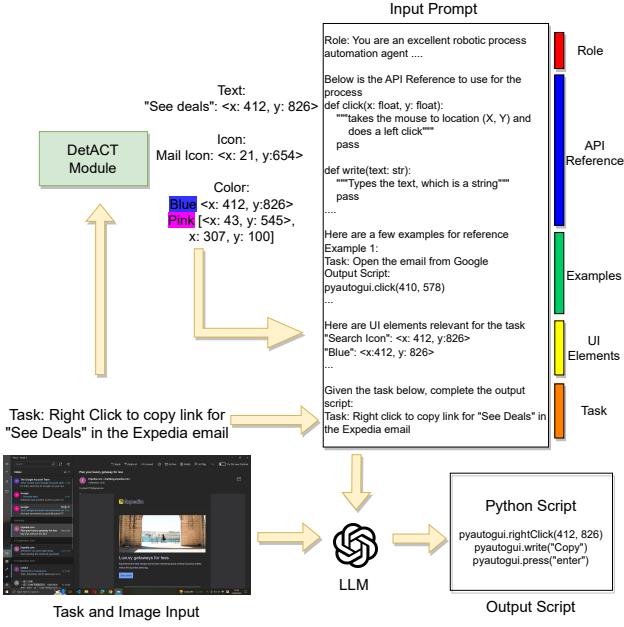
**Action Score =**

$$\frac{\sum_i \max \left( SeqScore_i - \sum_j (M_i^j + K_i^j + W_i^j), 0 \right)}{\sum_i SeqScore_i}$$

We report and discuss these metrics for all baseline models in Section 7.

## 5. DetACT: DETecting ACTions from UI

Understanding UI screens is crucial for multimodal computer tasks. Web-based agents typically use language-only inputs from the HTML DOM. This is insufficient for comprehending the full extent of an application UI, as many components may not be easily described with HTML code.



**Figure 4. Baseline Model Architecture.** Image and task descriptions are sent to DetACT module, which gives a filtered list of UI elements relevant to feed into the prompt along with the task. We also show the prompt structure used for action script generation. This structure is passed through the LLM (along with the image for multimodal LLM) to generate the automation script.

To address this, we propose DetACT, which allows us to convert images of UI layouts into structured code and text outputs for a downstream LLM. DetACT is a system comprised of three distinct modules: the text module, the icon module, and the color module.

1. **Text Extraction:** We use the EasyOCR model<sup>2</sup> to parse over the UI screens and collect all text-based elements. Along with the text, we also note the locations of each of these elements. This is depicted in Figure 3, along with a list of text elements found on the screen using the OCR Module. We segment and classify the different regions within the screenshot using the Segment Anything Model (SAM) [18]. From the outputs, we filter out the non-textual segments for our icon and color detection.
2. **Icon Module:** For matching with the appropriate icon, we use a pack of 1600 icons<sup>3</sup> as templates. Each of these icons is labeled with their appropriate functionality and is matched with the filtered outputs SAM [18]. For the similarity of the two images, we resize the reference icons and segmented region of interest (ROI) to the same size, and convert both images to grayscale. After this, we use the Structural Similarity Index (SSIM) [48], to find the closest match of the ROI to the icons in our set, and select the ones above the SSIM threshold of 0.95. As

<sup>2</sup><https://github.com/JaidedAI/EasyOCR>

<sup>3</sup><https://icomoon.io/>

seen in Figure 3, a few icons matched on the screen are *Globe* icon, *Calendar* icon, *Person* icon, and *Location* icon; each depicting a different use case.

3. **Color Module:** Finally, to place all segments of interest into appropriate buckets of colors, we average the RGB pixel values over the ROI and, based on that value, bucket them into different color categories. We categorize colors differently based on the human perspective of the ranges of each color. To avoid ambiguity, we consider eleven major colors, namely yellow, blue, green, red, pink, violet, white, black, orange, brown, and grey. We record the center of the element along with the color.

Once all the elements of each category are extracted with their coordinates, we then filter these UI elements by prompting GPT-4 [30]. We ensure that the elements selected are suited only for our task, for which we also provide the task description in our prompts along with the list of elements. Full details of the prompt are provided in the appendix section of the paper. As we observe in Figure 3, given an image from the Expedia application, and a task (“*Click on the Black Location icon and enter the destination as Paris.*”), the LLM filters out the elements to retain only “*Going To*”, “*Location Icon*”, and the *Black* colored elements from the screen. This is passed as input to the LLM or vision-language model backbone.

## 6. Baselines

To evaluate the performance of existing language model-based agents on *OmniACT*, we conduct experiments with both language-based and multimodal baselines. The DetACT module takes in image and text descriptions of the task and outputs the color, icon, and text-based signals. This is concatenated to the prompt for the LLM prompt-based baselines (see Figure 4). Every prompt starts with a role assignment [55], followed by the detailed API reference of the *PyAutoGUI* function set, along with a textual description of their function. We then add five in-context examples from the training set that most closely match the task (based on the cosine similarity of the MiniLM [46] embeddings of the reference task and the train examples). We add a list of UI elements filtered by the DetACT module to the prompt. Finally, we provide the rules with the task description. For multimodal baselines, we also pass the image pixels to the vision encoder. We report the results of several baselines:

- **Few-shot Generative LLM:** We experiment with models from LLaMA-2 [43], Vicuna-1.5 [7], CodeLLaMA-34B [34], Palmyra [42], and GPT [30] series. We use the prompts structure as shown in Figure 4 to prompt the model. For LLaMA and CodeLLaMa, we reduce the prompt length to 2000 tokens by removing outputs from the DetACT module with lower confidence, as we observed poor performance on longer prompts. For the other models, we allow prompts with up to 4000 token sizes.

Model	SS( $\uparrow$ )	$M_p$	$K_p$	$W_p$	AS( $\uparrow$ )
<b>Prompt based LLMs</b>					
LLaMA-7B [43]	4.12	1.24	1.83	0.57	0.48
Vicuna-7B [7]	3.88	1.17	1.51	0.43	0.77
LLaMA-13B [43]	4.80	1.32	0.93	0.93	1.62
Vicuna-13B [7]	5.44	1.65	0.94	1.06	1.78
Palmyra-Instruct-30B [41]	7.51	5.68	0.12	0.40	1.31
CodeLLaMA-34B [35]	10.09	2.99	2.71	0.66	3.72
Palmyra-X 43B [2]	11.20	3.12	3.02	2.12	2.94
GPT-3.5-turbo-0613 [29]	22.85	8.13	4.51	2.31	7.89
GPT-4 [30]	<b>32.75</b>	10.27	6.99	3.89	<b>11.60</b>
<b>Finetuned LLMs</b>					
LLaMA-13B FT	<b>8.92</b>	4.61	1.43	0.74	2.14
Vicuna-13B FT	8.78	4.12	1.31	0.63	<b>2.72</b>
<b>Multimodal LLMs</b>					
LLaVA-v1.5-7B [25]	13.23	4.73	1.24	1.44	5.82
LLaVA-v1.5-13B [24]	<b>20.56</b>	6.07	3.44	2.85	<b>8.19</b>
Human Performance	82.23	0.12	0.36	1.61	80.14

Table 4. Baseline Performance. (A) Prompt-only LLMs, (B) Fine Tuned LLMs, (C) Prompt-only Multimodal Models. The table represents the Sequence score (SS), click penalty ( $M_p$ ), Key penalty ( $K_p$ ), Write Penalty ( $W_p$ ), and Action Score (AS). The best results for the (SS) and (AS) are highlighted.

Model	Sequence Score ( $\uparrow$ )	Action Score ( $\uparrow$ )
GPT-4 [30]	36.42	12.77
GPT-4V [50]	<b>39.43</b>	<b>20.76</b>

Table 5. Results of GPT-4 and GPT-4V on a subset of 500 samples.

- **Finetuned Generative LLM:** We fine-tuned the LLaMA-13B model and Vicuna-13B using QLoRa [10] with rank 64 and scaling factor 16 for 300 steps to generate the code given screen description from the DetACT module and the instruction.
  - **Few-shot Generative Multimodal Models:** As **OmniACT** is predominantly multimodal, with a majority of tasks being visually grounded, we conduct experiments with large multimodal models. Given the limited research in this domain [47, 52], there is a scarcity of available multimodal models with significant size adept for this task. Here, we experiment with [24, 25], providing a similar prompt as well as the screen image.
- We report all results over the test set in Table 4.

## 7. Results and Analysis

As shown in Table 4, we experiment with three different categories of models, namely Prompt-based LLMs, Fine-tuned LLMs, and Prompt-based Multimodal Models. GPT-4 is the best-performing approach, scoring higher on the sequence score and invoking lower penalties on coordinate predicting and text input. For prompt-only LLMs, the GPT-3.5-turbo and GPT-4 models outperform the other LLM baselines, including the LLaMA [43] and Vicuna [7] models. We observe that CodeLLaMA-34B [35], which is trained for code generation, also achieves a higher performance than other models of the same size at predicting the action sequences.

Fine-tuned models also perform much better than their few-shot prompt-only counterparts. Fine-tuning substan-

tially improves LLaMA-13B’s sequence score (4.80 to 8.92) and action score (1.62 to 2.14), as well as the other metrics. Despite this, we observed that both, prompt-based LLMs and finetuned LLMs face severe mouse penalties, especially on click coordinates. This is because they rely solely on text-based signals.

To address this, we experiment with multimodal language models (Table 4). We observe that the coordinate prediction improves significantly when we provide the entire image as input to the multimodal LLM, as this enables it to fully utilize the screen representation. In addition to open sourced models, we also experiment with the GPT-4-vision-preview API [50] on a subset of 500 data points (due to cost overheads and OpenAI API request limits). Table 5 shows that GPT-4 Vision [50] outperforms GPT-4 significantly on the Action Score along with improving the sequence score, which we attribute to the strong reasoning abilities of GPT-4 coupled with the improved visual understanding capabilities of the GPT-4-vision-preview model [50]. These findings pave the way towards exciting new research directions on building multimodal models for long-horizon planning and code generation.

**Human performance over the task:** **OmniACT** consists of visually complicated tasks, and tests various types of computer skills. In order to get a gauge of how well humans perform, we collect evaluation data from human evaluators. We split the test set uniformly amongst 10 human evaluators, and provided them with the screenshot and task instruction. We record the actions taken by the annotators, and measure their performance on our predefined metrics (Table 4). We find that users generally exhibit a high level of proficiency when attempting most tasks for the first time. However, there are instances where users face difficulties in successfully completing certain tasks. These are due to factors including the user’s inability to fully comprehend the task, difficulties in grounding the task to the provided screenshot, or a lack of familiarity with the UI.

## 8. Conclusion and Future Work

Autonomous virtual agents offer the potential to automate routine tasks, benefiting users with limited technical expertise. To solve this task, we introduce **OmniACT**, a unique dataset of 9.8K human-labeled data points. **OmniACT** benchmarks autonomous agents across a range of tasks on web and desktop applications. LLM-based agents, like GPT-4, achieve a respectable action score of 11.6 on our dataset. However, **OmniACT** presents a challenge for the current state-of-the-art language and multimodal models. It provides a direction for future research on foundational multimodal models that seamlessly integrate language and visual understanding of computer screens and stands poised to drive the next wave of advancements in generalist autonomous agents offering omnipotent assistance to humans.

## References

- [1] Pyautogui: A cross-platform gui automation python module for human beings. <https://github.com/asweigart/pyautogui>, 2023. 2, 3
- [2] Waseem AlShikh, Manhal Daaboul, Kirk Goddard, Brock Imel, Kiran Kamble, Parikshith Kulkarni, and Melisa Russak. Becoming self-instruct: introducing early stopping criteria for minimal instruct tuning, 2023. 2, 8
- [3] Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, and Blaise Aguera y Arcas. Uibert: Learning generic multimodal representations for ui understanding, 2021. 2
- [4] Pratyay Banerjee, Shweta Mahajan, Kushal Arora, Chitta Baral, and Oriana Riva. Lexi: Self-supervised learning of the ui language, 2023. 2, 3
- [5] Andrea Burns, Deniz Arsan, Sanjna Agrawal, Ranjitha Kumar, Kate Saenko, and Bryan A Plummer. Mobile app tasks with iterative feedback (motif): Addressing task feasibility in interactive visual environments. 3
- [6] Xingyu Chen, Zihan Zhao, Lu Chen, Jiabao Ji, Danyang Zhang, Ao Luo, Yuxuan Xiong, and Kai Yu. Websrc: A dataset for web-based structural reading comprehension. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4173–4185, 2021. 3
- [7] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhang-hao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yong-hao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%\* chatgpt quality, 2023. 2, 7, 8
- [8] Biplob Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, page 845–854, New York, NY, USA, 2017. Association for Computing Machinery. 2
- [9] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *arXiv preprint arXiv:2306.06070*, 2023. 2, 3, 4, 5
- [10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. 2, 8
- [11] Hiroki Furuta, Ofir Nachum, Kuang-Huei Lee, Yutaka Matsuo, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models. *arXiv preprint arXiv:2305.11854*, 2023. 2
- [12] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962, 2023. 3
- [13] Izzeddin Gur, Ulrich Rueckert, Aleksandra Faust, and Dilek Hakkani-Tur. Learning to navigate the web. In *International Conference on Learning Representations*, 2018. 2
- [14] Zecheng He, Srinivas Sunkara, Xiaoxue Zang, Ying Xu, Lijuan Liu, Nevan Wichters, Gabriel Schubiner, Ruby Lee, Jindong Chen, and Blaise Aguera y Arcas. Actionbert: Leveraging user actions for semantic understanding of user interfaces, 2021. 2
- [15] Zhicheng Huang, Zhaoyang Zeng, Bei Liu, Dongmei Fu, and Jianlong Fu. Pixel-bert: Aligning image pixels with text by deep multi-modal transformers, 2020. 2
- [16] Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. A data-driven approach for learning to control computers. In *International Conference on Machine Learning*, pages 9466–9482. PMLR, 2022. 3
- [17] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491*, 2023. 3
- [18] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything, 2023. 7
- [19] Yann LeCun. A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. 2022. 2
- [20] Gang Li and Yang Li. Spotlight: Mobile ui understanding using vision-language models with a focus, 2023. 2
- [21] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. Mapping natural language instructions to mobile ui action sequences, 2020. 3
- [22] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. Widget captioning: Generating natural language description for mobile user interface elements, 2020. 2
- [23] Yang Li, Gang Li, Xin Zhou, Mostafa Dehghani, and Alexey Gritsenko. Vut: Versatile ui transformer for multi-modal multi-task user interface modeling, 2021. 2
- [24] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning, 2023. 8
- [25] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning, 2023. 8
- [26] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models, 2023. 3
- [27] Chenyang Lyu, Minghao Wu, Longyue Wang, Xinting Huang, Bingshuai Liu, Zefeng Du, Shuming Shi, and Zhaopeng Tu. Macaw-llm: Multi-modal language modeling with image, audio, video, and text integration. *arXiv preprint arXiv:2306.09093*, 2023. 3
- [28] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. 3
- [29] OpenAI. Introducing chatgpt, 2023. 8
- [30] OpenAI. Gpt-4 technical report, 2023. 2, 7, 8
- [31] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002. 5, 6

- [32] Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: A large-scale dataset for android device control, 2023. 3
- [33] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020. 5
- [34] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023. 7
- [35] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023. 8
- [36] Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina Toutanova. From pixels to ui actions: Learning to follow instructions via graphical user interfaces. *arXiv preprint arXiv:2306.00245*, 2023. 2
- [37] Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning*, pages 3135–3144. PMLR, 2017. 2, 3, 4
- [38] Abishek Sridhar, Robert Lo, Frank F Xu, Hao Zhu, and Shuyan Zhou. Hierarchical prompting assists large language model on web navigation. *arXiv preprint arXiv:2305.14257*, 2023. 3
- [39] Liangtai Sun, Xingyu Chen, Lu Chen, Tianle Dai, Zichen Zhu, and Kai Yu. Meta-gui: Towards multi-modal conversational agents on mobile gui. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 6699–6712, 2022. 3
- [40] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergrpt: Visual inference via python execution for reasoning, 2023. 3
- [41] Writer Engineering team. InstructPalmyra-30b : Instruct tuned Palmyra-Large model. <https://dev.writer.com>, 2023. 2, 8
- [42] Writer Engineering team. Palmyra-base Parameter Autoregressive Language Model. <https://dev.writer.com>, 2023. 7
- [43] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. 2, 7, 8
- [44] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. Screen2words: Automatic mobile ui summarization with multimodal learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 498–510, 2021. 2
- [45] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*, 2023. 2
- [46] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in Neural Information Processing Systems*, 33:5776–5788, 2020. 7
- [47] Xiao Wang, Guangyao Chen, Guangwu Qian, Pengcheng Gao, Xiao-Yong Wei, Yaowei Wang, Yonghong Tian, and Wen Gao. Large-scale multi-modal pre-trained models: A comprehensive survey, 2023. 8
- [48] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. 7
- [49] Nancy Xu, Sam Masling, Michael Du, Giovanni Campagna, Larry Heck, James Landay, and Monica Lam. Grounding open-domain instructions to automate web support tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1022–1032, 2021. 3
- [50] Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. The dawn of lmms: Preliminary explorations with gpt-4v (ision). *arXiv preprint arXiv:2309.17421*, 9, 2023. 2, 8
- [51] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022. 3
- [52] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. A survey on multimodal large language models, 2023. 8
- [53] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert, 2020. 5
- [54] Zhuosheng Zhang and Aston Zhang. You only look at screens: Multimodal chain-of-action agents, 2023. 3
- [55] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023. 7
- [56] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pre-trained models of code. *arXiv preprint arXiv:2302.05527*, 2023. 5
- [57] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023. 2, 3, 4

## Appendix

In this paper, we present a novel dataset **OmniACT** that aids in building more robust multimodal generalist autonomous agents for Desktop and Web. Along with this, we also propose a new continuous scale metric that allows better assessment of actions on a computer screen and the DetACT module which we integrate into multiple LLMs and (Vision Language Models) VLMs that can extract the useful features from the screen image and help us benchmark the dataset. We present the following items that give further insight into the dataset and experiments we performed:

- A** List of applications and websites
- B** Additional Quantitative Results
- C** Annotation Process
- D** Dataset and Metadata Format
- E** Parameters for Model Training
- F** Limitations and Broader Impact
- G** Ethics Statement
- H** Sample of task execution
- I** Prompts for DetACT and Baselines and Sample Responses

### A. List of applications and websites

The complete list of applications and websites chosen for the dataset are listed in table 6. We ensure that the data is equitably represented across all domains and operating systems. We choose desktop applications that are more commonly used for each of the operating systems. The commonly used applications we select are representative of eclectic demographics. We also select a few third-party applications that users commonly use on all operating systems. To avoid redundancy and ensure diversity in our tasks, we refrain from choosing the same set of applications for all three operating systems. For websites, we select our candidate set based on user intent. This is sub-categorized into six domains - (1) Shopping, (2) Entertainment, (3) Service, (4) Government, (5) Travel, and (6) Health.

### B. Additional Discussions

#### B.1. Screen Region Attention

We perform an in-depth analysis of our dataset to see where the coordinate predictions are gravitated. We find interesting insights about the parts of the screen that are attended towards. Most of the predictions for the desktop data are focused either on the center of the screen or the bottom corners. While most of the web interactions are pointing toward the bottom half of the screen, especially concentrating toward the corners. This is depicted in figure 5. This presents some exciting directions for training multimodal

models that will heavily rely on the screen image in addition to the elements extracted from DetACT module.

From the histograms at the top and right of the screens in figure 5, we notice that the dataset is not heavily biased towards any particular region of the screen. This implies that models that are likely to attend to only certain regions of the screen or predict the coordinates in the same region of the screen will not have a good performance over the dataset.

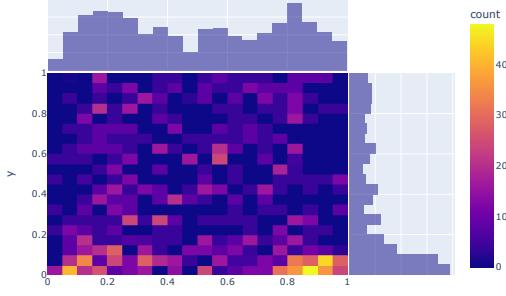
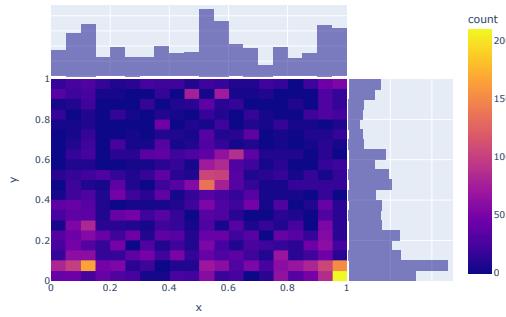


Figure 5. Screen region attention in output scripts for desktop and web data

#### B.2. Impact of DetACT Module

In this set of experiments we present the results without using our DetACT module and their comparison with the corresponding model with DetACT outputs. We can clearly observe in table 7 that even models like GPT-4 perform close to zero when not prompted by DetACT as they are only language-based. This highlights the importance of well-equipped screen parsing modules such as our DetACT framework that effectively extract multimodal information in the form of text, which is an essential cue for every model.

#### B.3. Sequence length versus Performance

We analyze the performance of predicting the sequence of actions as a function of sequence length. Specifically, we

Desktop					Websites		
MacOS			Linux	Windows			
Mail	Music	App Store	Audible	Grammarly	AMC	Coursera	NPS
Maps	News	Calculator	Camera	Outlook	Apartments	FlightAware	NY Gov
Message	Photos	Calendar	Desktop	Spotify	Aramex	Google Careers	Samsung
Prime	Preview	Clock	Files	WeChat	Armani	HealthLine	Trip Advisor
Terminal	Reminder	Expedia	Settings	Calendar	Asics	Hertz	UHaul
Weather	Stocks	Finder	Todo	Files	AT&T	Indeed	Udemy
Zoom	System Preferences	iBooks	Text Editor	Settings	BestBuy	Instacart	United Airlines
Desktop			VLC	Windows Store	Booking	Mayo Clinic	UPS
					Cmu.edu	Mint Mobile	Walmart

Table 6. List of applications for desktop and web in OmniACT

Model	Sequence Score	Action Score
GPT-4 (w/o DetACT)	31.87	0.32
GPT-4 + DetACT	32.75	11.60

Table 7. Performance of DetACT Module with GPT-4 Series of Models

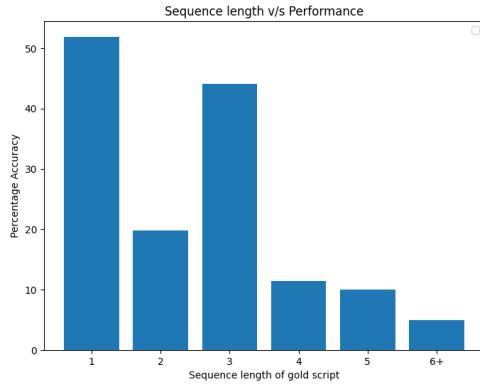


Figure 6. Sequence prediction accuracy of GPT4 on the test set. The performance decreases as the sequence length increases.

want to check whether it becomes difficult for models to predict the sequence correctly when the target sequence is longer. Figure 6 shows the accuracy of sequence prediction as a function of sequence length for GPT-4 on the test set. Only when the entire sequence of actions predicted by the model is the same as the gold sequence of actions, do we give the score as 1. We report the accuracy percentage in the table. From table 6, we observe that the model performance is better for shorter sequences and gets worse as the sequence length increases.

## C. Annotation Process

### C.1. Creating the Bounding Boxes

To create the bounding boxes, we use two different techniques for desktop and web data. For desktop data, we

use a custom PyQt5 tool that helps us create the bounding boxes using a click-and-drag mechanism. This enhances the bounding box creation process, making it more intuitive and user-friendly. It also enables us to encompass interactive regions of UI elements. For instance, when clicking on a search bar, users can input text, while clicking on a search bar icon (such as a magnifying glass) does not activate the typing area. Next, for the website, we use the DOM from HTML and draw bounding boxes using JavaScript. Both these processes ensure a high degree of accuracy and are done by the authors of the paper. Samples for each of them are shown in figure 7. After the bounding boxes are established, the authors exchange their work and review the boxes created by each other.

### C.2. Annotator Guidelines

For labeling the bounding boxes for both desktop applications and websites, we leverage MTurk. This is depicted in figure 8. MTurk workers are present with a single bounding box on the screen removing the other boxes and are then asked to label the box based on the functionality of the UI element in context to the screen. They are asked to come up with less than a 5-word description or label for each box. Every box is doubly annotated keeping the more appropriate label during the final filtering. Every MTurk worker is paid at an hourly rate of \$20, which is well above Pennsylvania's minimum wage rate.

Next for formulating the tasks, we recruit student volunteers who have basic knowledge of Python and coding. We train every student volunteer about the *PyAutoGUI* usage in a thirty-minute session before starting the task formulation process. Every student is then handed over a sample of tasks that can be created using that screen and is asked to come up with more tasks along with their scripts, which are only a few lines. Every volunteer is asked to come up with as many tasks as possible in one hour along with its linguistic variations to ensure diversity in the language of the tasks. Every volunteer is paid an hourly wage rate of \$25, which again surpasses the state's minimum rate.

We build scripts to take in the tasks generated by the stu-

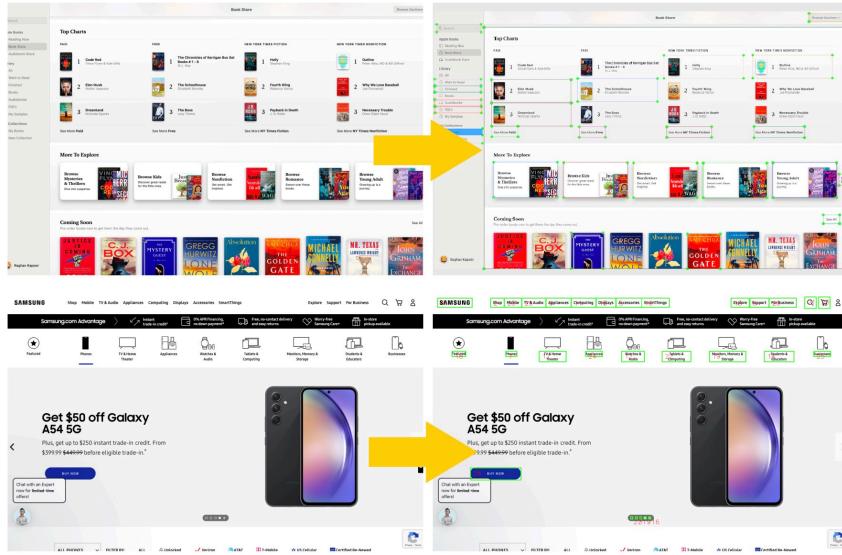


Figure 7. Annotation Process for creating bounding boxes for desktop applications and websites.

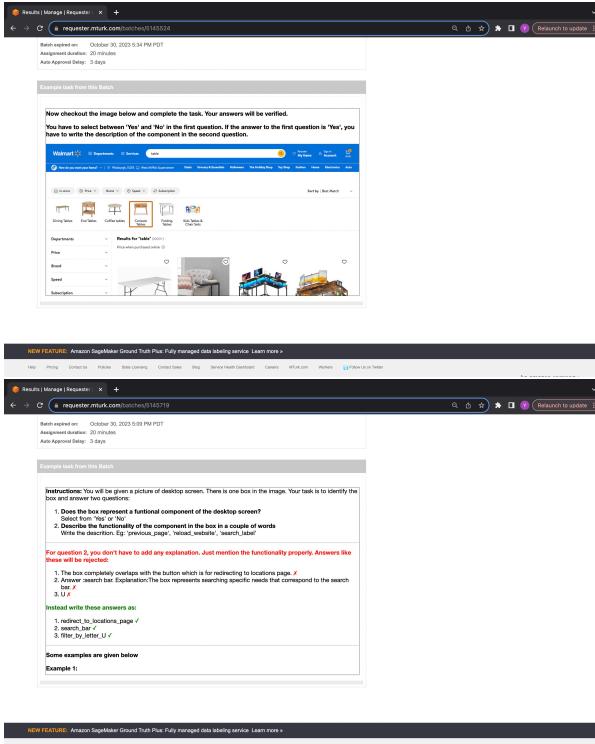


Figure 8. Screenshots from MTurk Portal that depict the labeling process for all boxes.

dents and run over the screens to ensure the tasks are executable. Once done, the authors perform a final pass to keep only the high-quality tasks.

### C.3. Human Feedback and Evaluations

To evaluate the human baselines, we again recruit student volunteers. This time the volunteers are not given the reference output script and only a pair of an image and the task description. The students are asked to then come up with an automation script using a few samples as references. The volunteers use their experience and intuition to comprehend the screen and the functionalities and write scripts based on that. Here the student worker is compensated at an hourly rate of \$25, well exceeding the state's minimum wage.

### D. Dataset and Metadata Format

The dataset consists of three files, one for each split. Every file has a list of task numbers for that particular split. Every task directory consists of (1) *image.png* file, which is a screenshot of the screen, and (2) *task.txt* file which contains the task description and the ground truth output script. We also include another file, (3) *box.json*, which is part of the metadata and is meant to be only used for running the evaluations that require the coordinates of the boxes and not for performing the task or aiding the model in any way.

### E. Parameters for Model Training

We enlist the model specifications for models we run for DetACT filtering and baselines using DetACT in table 8.

### F. Broader Impact

The broader impact of our intricately curated dataset is multifaceted. Firstly, our dataset holds the potential to empower the technologically illiterate, providing them with a more

	<b>Model</b>	<b>Hyperparameters</b>
DetACT Filtering	GPT-4	temperature = 0.1
LLM Prompt Only Baselines	LLaMA-7B	max_new_tokens = 400,
	Vicuna-7B	temperature = 0.8, repetition_penalty = 1.1
	LLaMA-13B	max_new_tokens = 400,
	Vicuna-13B	temperature = 0.8
	Palmyra-Instruct 30 B	temperature = 0.7
	CodeLLaMA-34B	max_new_tokens = 400, temperature = 0.7
LLM Fine-tuned baselines	Palmyra-X 43B	temperature = 0.7
	GPT-3.5-turbo	temperature = 0.3
	GPT-4	temperature = 0.3
LLM Fine-tuned baselines	LLaMA-13B FT	LoRA rank = 64, scaling factor = 16,
	Vicuna-13B FT	LoRA dropout = 0.05, batch size = 4, learning rate = 5e-5
Multimodal Model Prompt Only Baselines	LLaVA-v1.5-7B LLaVA-v1.5-13B	temperature = 0.2, max_new_tokens = 512

Table 8. Hyperparameters used for each of the models for DetACT filtering and Baselines.

automated and accessible way to navigate and operate computers.

Additionally, our dataset opens avenues for groundbreaking UI grounding research, enabling a deeper understanding of user interactions at the operating system level. This, in turn, contributes to advancing the field of artificial intelligence (AI) by expanding the scope of AI to encompass OS-level interactions. The dataset’s rich content further facilitates the enhancement of large language models (LLMs), broadening their skill sets and improving their capabilities in comprehending and responding to diverse user inputs.

This dataset serves as a foundation for the development of multimodal agents capable of operating seamlessly across various screens and executing tasks with long-term horizons. This versatility in application positions the dataset as a valuable resource for the creation of innovative AI-driven solutions that transcend traditional limitations.

An additional and impactful use case involves building assistive tools for handicapped individuals who may face challenges in operating computer systems or performing routine tasks that involve human-computer interaction. By tailoring technologies to meet the needs of individuals with varying abilities, our dataset contributes to the development of assistive technologies that enhance accessibility and improve the quality of life for a diverse range of users.

In summary, the broader impact of our meticulously curated dataset manifests in its potential to drive advancements in technology accessibility, UI grounding research, AI capabilities, and assistive tools, ultimately fostering a more inclusive and empowered technological landscape.

## G. Ethics Statement

This work introduces a valuable dataset, yet we recognize a few limitations that exist. State-of-the-art models like GPT-4, may exhibit susceptibility to hallucinations and bias towards specific data types, hindering broad applicability.

Reliance on closed models like GPT-4V poses integration challenges due to high costs and time constraints in real-world systems. Despite efforts for equal representation and data collection without personal information, biases may be introduced as the dataset is exclusively in English, and the human-curated content may have temporal biases.

## H. Sample of task execution

We present figures 9 containing screenshots and corresponding output scripts for a selected task. In figure 9, the task is taken from a MacOS native application, *Books*. Here we have a book chapter opened along with the task description to highlight the last line in green color and scroll to the next page. Followed by the task are descriptive screenshots representing every action in the sequence along with its output.

## I. Prompts for DetACT and Baselines and Sample Responses

In the following pages, we present the full length prompts that we use for the following tasks:

- Listing 1: Filtering UI elements in DetACT Module
- Listing 2: Running baselines models using DetACT



---

**Listing 1. Prompt for Filtering UI elements in DetACT Module**

---

Given below are the UI elements extracted from the screen. You have to filter the elements for this UI screen that are relevant for carrying out the task given below.

Make sure to filter the UI elements that may be helpful to carry out the task and mention the element description and the corresponding coordinates for the task.

Format for every element would be in the form of (Element Label, X-Coordinate, Y-Coordinate).

[IMPORTANT!!] Only remove the elements if you are sure that they will not help the task.

[IMPORTANT!!] Follow the output structure strictly as given in the below example and output nothing else other than the required output.

Sample Task:

{SAMPLE\_TASK}

Sample UI Elements:

{LIST\_OF\_SAMPLE\_UI\_ELEMENTS}

Sample Filtered UI Elements:

{LIST\_OF\_SAMPLE\_FILTERED\_UI\_ELEMENTS}

Given Task:

{TASK}

Given UI Elements:

{LIST\_OF\_UI\_ELEMENTS}

---

Listing 2. Prompt for baselines models using DetACT outputs

You are an excellent robotic process automation agent who needs to generate a pyautogui script for the tasks given to you. You will be given UI elements on the screen that you can interact with as a list of tuples, where each tuple will be of the form [UI Text, X coordinate, Y coordinate]. You will also be given an example to help with the format of the script that needs to be generated.

[IMPORTANT!!!] Stick to the format of the Output scripts in the example  
[IMPORTANT!!!] Use only the functions from API docs  
[IMPORTANT!!!] Follow the Output format strictly. Only write the script and nothing else.

Here is the API reference for generating script:

```
def click(x=moveToX, y=moveToY):
    """takes the mouse to location (moveToX, moveToY) and does a left click
Example:
High Level Goal: Click at co-ordinate (150, 230).
Python script:
import pyautogui
pyautogui.click(150, 230)
"""
pass

def rightClick(x=moveToX, y=moveToY):
    """takes the mouse to location (moveToX, moveToY) and does a right click
Example:
High Level Goal: Right click at co-ordinate (350, 680).
Python script:
import pyautogui
pyautogui.rightClick(350, 680)
"""
pass

def doubleClick(x=moveToX, y=moveToY):
    """takes the mouse to location (moveToX, moveToY) and does a right click
Example:
High Level Goal: Right click at co-ordinate (350, 680).
Python script:
import pyautogui
pyautogui.rightClick(350, 680)
"""
pass

def scroll(clicks=amount_to_scroll):
    """scrolls the window that has mouse pointer by float value (amount_to_scroll)
Example:
High Level Goal: Scroll screen by (30).
Python script:
import pyautogui
pyautogui.scroll(30)
"""
pass

def hscroll(clicks=amount_to_scroll):
    """scrolls the window that has mouse pointer horizontally by float value
(amount_to_scroll)
Example:
High Level Goal: Scroll screen horizontally by (30).
```

```

Python script:
import pyautogui
pyautogui.hscroll(30)
"""
pass

def dragTo(x=moveToX, y=moveToY, button=holdButton):
    """drags the mouse to (moveToX, moveToY) with (holdButton) pressed. holdButton can be
    'left', 'middle' or 'right'.
Example:
High Level Goal: drag the screen from current position to (450, 600) with left click of
the mouse.
Python script:
import pyautogui
pyautogui.dragTo(450, 600, 'left')
"""
pass

def pyautogui.moveTo(x=moveToX, y=moveToY):
    """take the mouse pointer to (moveToX, moveToY)
Example:
High Level Goal: hover the mouse pointer to (450, 600).
Python script:
import pyautogui
pyautogui.moveTo(450, 600)
"""
pass

def write(str=stringType, interval=secs_between_keys):
    """writes the string wherever keyboard cursor is at the function calling time with
    (secs_between_keys) seconds between characters
Example:
High Level Goal: Write "Hello world" with 0.1 seconds rate
Python script:
import pyautogui
pyautogui.write("Hello world", 0.1)
"""
pass

def press(str=string_to_type):
    """simulate pressing a key down and then releasing it up. Sample keys include 'enter',
    'shift', arrow keys, 'f1'.
Example:
High Level Goal: Press enter key now
Python script:
import pyautogui
pyautogui.press("enter")
"""
pass

def hotkey(*args = list_of_hotkey):
    """Keyboard hotkeys like Ctrl-S or Ctrl-Shift-1 can be done by passing a list of key
    names to hotkey(). Multiple keys can be pressed together with a hotkey.
Example:
High Level Goal: Use Ctrl and V to paste from clipboard
Python script:
import pyautogui
pyautogui.hotkey("ctrl", "v")

```

```
"""
pass
```

Here are a few examples for generating the script and format to be followed:

Example 1:  
{RETREIVED\_EXAMPLE\_1}

Example 2:  
{RETREIVED\_EXAMPLE\_2}

Example 3:  
{RETREIVED\_EXAMPLE\_3}

Example 4:  
{RETREIVED\_EXAMPLE\_4}

Example 5:  
{RETREIVED\_EXAMPLE\_5}

Here are the Textual Elements found on the screen along with their coordinates:  
{TEXTUAL\_ELEMENTS\_FROM\_DETACT}

Here are the Icon/Image Elements found on the screen along with their coordinates:  
{ICON\_ELEMENTS\_FROM\_DETACT}

Here are the Color Elements found on the screen along with their coordinates:  
{COLOR\_ELEMENTS\_FROM\_DETACT}

Based on the list of UI elements and the examples above, generate the pyautogui script for  
the following task:

---

```
{TASK}
```