You Only Look at Screens: Multimodal Chain-of-Action Agents

Zhuosheng Zhang¹*, Aston Zhang²*
¹ School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University
² GenAI, Meta
zhangzs@sjtu.edu.cn, az@astonzhang.com

Abstract
Autonomous graphical user interface (GUI) agents aim to facilitate task automation by interacting with the user interface without manual intervention. Recent studies have investigated eliciting the capabilities of large language models (LLMs) for effective engagement in diverse environments. To align with the input-output requirement of LLMs, most existing approaches are developed under a sandbox setting where they rely on external tools and application-specific APIs to parse the environment into textual elements and interpret the predicted actions. Consequently, those approaches often grapple with inference inefficiency and error propagation risks. To mitigate the challenges, we introduce Auto-GUI, a multimodal solution that directly interacts with the interface, bypassing the need for environment parsing or reliance on application-dependent APIs. Moreover, we propose a chain-of-action technique—leveraging a series of intermediate previous action histories and future action plans—to help the agent decide what action to execute. We evaluate our approach on a new device-control benchmark AITW with 30K unique instructions, spanning multi-step tasks such as application operation, web searching, and web shopping. Experimental results show that Auto-GUI achieves state-of-the-art performance with an action type prediction accuracy of 90% and an overall action success rate of 74%. Code is publicly available at https://github.com/cooelf/Auto-GUI.

1 Introduction
Building intelligent autonomous agents that are capable of task planning, decision making, and action execution in a particular environment is a longstanding goal of artificial intelligence (AI) (Searle, 1969; Wooldridge and Jennings, 1995; Maes, 1995; Hendler, 1999). The advent of large language models (LLMs) (Brown et al., 2020; Chowdhery et al., 2022; OpenAI, 2023) has flourished promising opportunities for developing autonomous agents to assist users in completing tasks in distinct environments such as operation systems, specific applications, and web browsers (Adept, 2022; Rawles et al., 2023; Liu et al., 2023; Zhou et al., 2023; Wang et al., 2023c; Koh et al., 2024; Gao et al., 2023; Yang et al., 2023).

Recent studies have explored prompt engineering (Richards, 2023; Nakajima, 2023; Reworkd, 2023; Sumers et al., 2023; Liu et al., 2023) and fine-tuning techniques (Rawles et al., 2023; Wen et al., 2023; Sun et al., 2022) to elicit the capability of language models to execute actions in interactive environments. However, there are at least two major challenges that have limited real-world applications of autonomous agents.

First, existing approaches commonly rely on external tools such as optical character recognition (OCR) and icon detectors (Zhang et al., 2021; Sunkara et al., 2022) to parse the environment into textual elements (e.g., HTML layouts) as inputs to a language model (Figure 1(a)) (Rawles et al., 2023). On the one hand, the parsed elements generate lengthy inputs, thus leading to inference inefficiency. Since computational latency is a key measure in deployment, using lengthy inputs would increase inference cost and may even exceed the input length limit of the language model. On the other hand, parsing the visual environment into textual elements may also be prone to error propagation or information loss because parsing mistakes are inevitable using external tools.

Second, most existing approaches are under the sandbox setting that requires accessing internal APIs to interact with the environment (Zhou et al., 2023; Gur et al., 2023), e.g., using a JavaScript element selection on a webpage or a Python interpreter to execute actions. However in practice, the API interface is often inaccessible in third-party applications (Apps).
These challenges have motivated more advanced techniques that are capable of first principles thinking (Aristotle; Irwin, 1989)—allowing direct interactions on the screen without needing access to intermediate environment parsing or interval application-dependent APIs (Figure 1(b)). To address the challenges, we introduce Auto-GUI, a multimodal approach that directly interacts with the graphical user interfaces (GUIs). To further strengthen the agent’s action prediction capability, we propose a novel chain-of-action technique, where a chain of action is a series of intermediate previous action histories and future action plans.

We evaluate Auto-GUI on a new device-control benchmark AITW (Rawles et al., 2023) with 30K unique instructions, spanning multi-step tasks of application operation, web searching, and web shopping. Experimental results show that Auto-GUI achieves state-of-the-art performance with an action type prediction accuracy of 90% and an action success rate of 74%. Notably, Auto-GUI is able to infer an action in less than one second.

2 Related Work
Our work falls into the field of language agents. This section will first review the recent progress in building language agents and then discuss the approaches to conduct user interface control with language agents.

2.1 Language Agents
Language agents refer to those agents that can follow user instructions and interact with environments to complete tasks. Such agents expand the landscape of language models to compete in specific fields, including application operation, web searching, and web shopping. There are two popular types of language agents, autonomous agents and communicative agents. Autonomous agents aim to assist humans to achieve specific goals in the real world. Typical examples of autonomous agents are AutoGPT (Richards, 2023), BabyAGI (Nakajima, 2023), and AgentGPT (Reworkd, 2023). In contrast, communicative agents are personalized and socialized agents (Park et al., 2023; Wang et al., 2023b; Zhu et al., 2023; Hong et al., 2023a) with human behaviors that can communicate and collaborate with each other. They are often deployed in immersive environments.

Inspired by the potential in real-world applications, this work focuses on autonomous agents, es-
especially those working in mobile devices. We aim to assist users by completing multi-step tasks (e.g., manipulating Apps, web shopping, and question answering) without any manual intervention. Given a user instruction in natural language, the agent is required to interpret the instruction and execute actions by directly controlling its user interface. Due to the requirement in real-world applications, the agent is expected to be both effective and efficient.

2.2 GUI Control with Natural Language

Recently, LLMs have shown promise in building autonomous GUI agents with abilities of instruction following (Sanh et al., 2022; Taori et al., 2023b; Chiang et al., 2023) and chain-of-thought (CoT) prompting (Nye et al., 2022; Wei et al., 2022). Especially, CoT prompting (Wei et al., 2022; Kojima et al., 2022; Zhang et al., 2023a) elicit LLMs’ capacities of step-by-step planning, decision making, and action execution. Those capacities have been shown to be effective in GUI control tasks (Rawles et al., 2023).

However, the task environments are GUIs instead of natural language that LLMs can process directly. Therefore, the GUI states and actions are required to be converted to textual formats to be applicable to LLMs. For example, it is feasible to parse the GUI screens by icon recognition and OCR (Zhang et al., 2021; Sunkara et al., 2022; Song et al., 2023) and organize the parsed elements into HTML layouts. As a compromise, existing approaches are restricted in a sandbox setting where they rely on external tools (Rawles et al., 2023; Wen et al., 2023) and application-specific APIs (Zhou et al., 2023; Gur et al., 2023) for environment parsing and action interpretation; thus, commonly suffer from inefficiency and error propagation. Although there are studies that have considered multimodal architecture for processing inputs in different modalities (Sun et al., 2022; Yan et al., 2023), those studies still rely on fine-grained environment parsing to ensure competitive performance. As GUI tasks have shown prerequisites to fine-grained grounding to the GUI contents, more recent concurrent studies (Cheng et al., 2024; Hong et al., 2023b) have explored GUI grounding pre-training to improve the agent’s performance.

In contrast to the studies above, this work is established upon first principles thinking, which directly reads GUI without additional environment parsing and provides the action (e.g., action type, gesture coordinate, and typed text) that can be efficiently executed without needing any extra APIs. Concretely, our approach diverges from existing studies in several key aspects.

(i) Conceptually, Auto-GUI represents a novel approach as a multimodal agent designed for autonomous GUI control, enabling direct interaction with screens. This approach effectively bypasses the need for environment parsing and access to application-specific APIs, thus enhancing adaptability and versatility.

(ii) Methodologically, our proposal introduces a chain-of-action technique. This method capitalizes on previously executed actions and future action plans to inform the decision-making process of the agent at each step.

(iii) As a scientific contribution, our paper ar-
ticulates that a unified multimodal model out of first principles thinking can serve as a strong autonomous agent evidenced by state-of-the-art performance. In addition, we facilitate a series of studies to understand the contributing factors regarding the vision features and model scale.

3 Methodology

In this section, we will first introduce the basic concepts for the GUI control task and then describe the design of our proposed Auto-GUI framework.

3.1 Problem Formalization

Given a user instruction (also known as a goal), the agent needs to complete the task with multiple steps of interactions. The entire process is called an episode, which is composed of a series of screens. For each step in the episode, the agent will be provided with a screenshot, and the agent is required to predict the action until the task is complete. Detailed examples can be found in Appendix A.1.

3.2 Framework Overview

Auto-GUI is a multimodal agent that decides what action to take given the input screenshot and a user instruction. To empower the agent’s decision-making capability, we introduce a chain-of-action approach by leveraging a series of intermediate previous action histories and future action plans to predict actions.

The model architecture of Auto-GUI is illustrated in Figure 2. On a high level, Auto-GUI consists of three stages. First, we acquire encoded features from both vision and language inputs. Specifically, the vision input, i.e., a screenshot, is encoded by a frozen vision encoder. Meanwhile, the language input, consisting of the goal and a chain of previous action histories—each history contains a tuple {action type, touch point, lift point, and typed text}, is encoded by a language encoder. Second, the encoded vision and language representations are integrated by a self-attention module. Third, the fused representation is fed to the decoder to generate a chain of future action plans (i.e., action types to execute in future steps) followed by action prediction. A chain of action consists of two parts in the procedure above: a chain of previous action histories on the input side and a chain of future action plans on the output side. In the following, we describe the entire procedure in detail.

Encoding Suppose that an episode consists of \( k \) steps of interactions. Given a screenshot \( X_{\text{screen}} \in \mathbb{R}^{h \times w \times 3} \) with height \( h \) and width \( w \) at step \( t \in [1, k] \), we first feed it to a frozen image encoder (e.g., BLIP-2 (Li et al., 2023)) and extract vision features \( H_{\text{screen}} \in \mathbb{R}^{1 \times d_s} \), where \( d_s \) is the dimension of the vision features. Additionally, we leverage a language encoder to extract the language features \( H_{\text{language}} \in \mathbb{R}^{n \times d_l} \) of the input goal \( X_{\text{goal}} \) where \( n \) is the number of tokens and \( d_l \) is the dimension of the language features. If \( t > 1 \), there will be a chain-of-action history already executed before step \( t \). We denote the chain of action histories as \( X_{\text{history}} = [m_1, \ldots, m_t] \) where \( m_t \) contains a tuple of action type, touch point, lift point, and typed text. Otherwise, if \( t = 1 \), \( X_{\text{history}} \) will be set empty:

\[
X_{\text{history}} = \begin{cases} [m_1, \ldots, m_t], & \text{if } t > 1 \\ \langle\text{empty}\rangle, & \text{otherwise} \end{cases}
\]

We concatenate \( X_{\text{goal}} \) and \( X_{\text{history}} \) as the input to the language encoder: \( X_{\text{language}} = \{X_{\text{goal}}, X_{\text{history}}\} \).

Then, we obtain the encoded representations of the vision and language inputs as follows:

\[
H_{\text{screen}} = \text{VisionExtractor}(X_{\text{screen}}),
\]

\[
H'_{\text{screen}} = W H_{\text{screen}},
\]

\[
H_{\text{language}} = \text{LanguageEncoder}(X_{\text{language}}),
\]

where \( W \) is a trainable projection matrix to convert \( H_{\text{screen}} \) into the same dimensionality as \( H_{\text{language}} \).

Interaction We correlate \( H'_{\text{screen}} \) and \( H_{\text{language}} \) with a single-head self-attention network (Vaswani et al., 2017),\(^2\) where the query (\( Q \)), key (\( K \)), and value (\( V \)) are \( H_{\text{language}} \), \( H'_{\text{screen}} \), and \( H_{\text{screen}} \), respectively. The attention output \( H_{\text{screen}}^{\text{attn}} \in \mathbb{R}^{n \times d} \) is defined as: \( H_{\text{screen}}^{\text{attn}} = \text{Softmax}(Q K^T) V \), where \( d_k \) is the same as the dimension of \( H_{\text{language}} \) because a single head is used.

Then, a gated fusion (Zhang et al., 2020; Wu et al., 2021; Zhang et al., 2023b) is adopted to fuse
We have the fused output $H_{\text{fuse}} \in \mathbb{R}^n \times d$ by:

$$
\begin{align*}
\lambda & = \text{Sigmoid}(W_l H_{\text{language}} + W_v H_{\text{attn}}), \\
H_{\text{fuse}} & = (1 - \lambda) \cdot H_{\text{language}} + \lambda \cdot H_{\text{attn}},
\end{align*}
$$

where $W_l$ and $W_v$ are learnable parameters.

**Decoding** The fused representation $H_{\text{fuse}}$ is fed to a Transformer decoder to generate the target predictions in a string format. The target predictions consist of a chain of future action plans $Y_{\text{plan}}$ and the current action prediction $Y_{\text{action}}$ separated by specific prompts: $\{\text{Action Plan: } Y_{\text{plan}}, \text{ Action Decision: } Y_{\text{action}}\}$. Concretely, $Y_{\text{plan}}$ is a chain of action types to execute in future steps: $Y_{\text{plan}} = \{\text{action_type}_1, \ldots, \text{action_type}_K\}$. $Y_{\text{action}}$ contains four components: $Y_{\text{action}} = \{\text{“action_type”: }<\text{action_type}>, \text{“touch_point”: }<\text{touch_point}>, \text{“lift_point”: }<\text{lift_point}>, \text{“typed_text”: }<\text{typed_text}>\}$. These four components will be explained as follows.

### 3.3 Coordinate Normalization

Recall that a target action consists of four components: action type, touch point, lift point, and typed text. We consider six action types: **dual-point gesture**, type, **go_back**, **go_home**, enter, and **status_complete**. A dual-point gesture comprises a touch point and a lift point with $[y, x]$ coordinates. The gesture actions ensure a flexible action space and can represent clicks and scrolls at arbitrary locations. For example, a gesture action {“touch_point”: [0.7761, 0.7089], “lift_point”: [0.7761, 0.7089]} means clicking at the coordinate $[0.7761, 0.7089]$, while a gesture action {“touch_point”: [0.1898, 0.4477], “lift_point”: [0.8242, 0.4077]} means scrolling down. A type action means typing a text and the text is placed in the <typed_text> field. The other action types, i.e., **go_back**, **go_home**, enter, and status_complete are system actions, whose corresponding <touch_point>, <lift_point> fields are filled with -1, and the <typed_text> is empty.

We observe that high-precision coordinates are not necessary for representing a click or scroll action. Therefore, we apply normalized values of the coordinates, which helps accelerate convergence and mitigate the ambiguity of coordinates. The normalization is applied to click and scroll actions. For click actions, we keep four decimal places. For scroll actions, we first determine the scroll direction with the touch and lift points. Then, we transform the touch and lift points into fixed directional coordinates as follows: “up”: $[[0.8, 0.5], [0.2, 0.5]]$, “down”: $[[0.2, 0.5], [0.8, 0.5]]$, “left”: $[[0.5, 0.8], [0.5, 0.2]]$, “right”: $[[0.5, 0.2], [0.5, 0.8]]$, where $[\cdot, \cdot]$ consists of the touch point and lift point in the first $\cdot$ and second $\cdot$. We provide examples of target actions in Appendix B.1.

### 4 Experiments

#### 4.1 Dataset

We use the AITW benchmark dataset (Rawles et al., 2023). AITW is a large-scale GUI control benchmark dataset containing natural language instructions, screenshots, and actions. There are $715K$ episodes spanning $30K$ unique instructions, covering diverse multi-step tasks such as application operation, web searching, and web shopping, on over 350 Apps and websites. This dataset covers various device types and operation systems in varying screen resolutions to ensure generality. There are five subsets in the benchmark dataset, namely, General, Install, GoogleApps, Single, and Web-Shopping.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Episodes</th>
<th>Screens</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>9,476</td>
<td>85,413</td>
<td>545</td>
</tr>
<tr>
<td>Install</td>
<td>25,760</td>
<td>250,058</td>
<td>688</td>
</tr>
<tr>
<td>GoogleApps</td>
<td>625,542</td>
<td>4,903,601</td>
<td>306</td>
</tr>
<tr>
<td>Single</td>
<td>26,303</td>
<td>85,668</td>
<td>15,366</td>
</tr>
<tr>
<td>WebShopping</td>
<td>28,061</td>
<td>365,253</td>
<td>13,473</td>
</tr>
</tbody>
</table>

Table 1: Dataset statistics.

Table 1 presents the data statistics of the AITW dataset. Each subset is split episode-wise into a training, validation, and test set (80/10/10%). More details of the subsets can be found in Appendix A.2.

#### 4.2 Baselines

We adopt three types of baselines, allowing for a comprehensive comparison with our approach. The baselines encompass the in-context earning (ICL) and fine-tuning paradigms. They are based on various backbone models of different sizes.

(i) **In-context Learning LLMs.** Few-shot PaLM 2, ChatGPT (turbo-3.5) are adopted. Following prior studies (Rawles et al., 2023; Wang et al., 2023a), we feed the LLM a textual description

---

3We follow the $[y, x]$ coordinate format in AITW.

4The numbers are relative scales and they generalize to different screen sizes.
of the screen and a user instruction. The screen is formatted as an HTML syntax, providing the information of GUI elements derived from OCR detection and icon detection from external tools (Rawles et al., 2023). The model is required to predict an action among pre-defined actions. In addition, we report the results of the multimodal GPT-4V by taking the vision image and action history as the input based on Yan et al. (2023).

(ii) Fine-tuned LLMs. We adopt Llama-2-7B (Touvron et al., 2023) as the baseline and fine-tune it with LoRA. We feed the model with the user instruction and the screen descriptions in HTML syntax (the same as in-context learning LLMs). The model is expected to predict the action in the same output format as in-context learning LLMs.

(iii) Specialized GUI Agent. We adopted the Behavioural Cloning (BC) agent, which reported the state-of-the-art performance in Rawles et al. (2023). BC is a Transformer-based architecture that takes a task instruction, the current screen, and a stacked history of screen observations and actions as input. All the embedded representations are fused to predict the action by a decoder. There are two BC variants, BC-single and BC-history, depending on whether the model takes the screen-action history as input.

More detailed implementation of the baselines can be found in Appendix B.2.

4.3 Evaluation Measures

We compute the screen-wise action matching score as the main evaluation measure, defined as the number of correct actions divided by the episode length (i.e., the number of screenshots). A predicted action is considered correct if the action type and dual-point gesture match the gold ones. As we described in Section 3.3, the gesture actions can represent the click actions and scroll actions at arbitrary locations. Following Rawles et al. (2023), a click action is considered correct if its touch point and lift point fall within a 14% screen distance from the gold gestures or occur within the same detected bounding box with the gold gestures. A scroll action is considered correct if it has the same scroll axis as the gold gesture.

The screen-wise action matching score has been shown to correlate with the task complete score estimated by human evaluations (Rawles et al., 2023) and is appropriate to measure the action success rate for user instructions. Besides the overall matching score, we will also compare the click region accuracy, scroll direction accuracy, action type accuracy (e.g., clicking, scrolling, typing, etc), and typed text accuracy for a more comprehensive reference (Section 5.1).

The evaluation criteria above apply to the BC baselines and our Auto-GUI. For the LLMs, they can only click on detected GUI elements, rather than clicking at arbitrary locations. Therefore, we consider if the clicked GUI element is matched for click actions instead of comparing dual-point gestures for LLMs.

4.4 Implementation Details

We adopt the encoder-decoder architecture (Raffel et al., 2020) under small (60M), base (200M) and large (700M) settings in our framework. We apply FLAN-Alpaca to initialize our model weights. The vision features are obtained by the frozen BLIP-2 encoder (Li et al., 2023) (version: blip2_t5_instruct). We fine-tune the models up to 10 epochs, with a learning rate of 1e-4. The maximum input sequence length is 512. The batch size is 4. Our experiments are run on 8 NVIDIA Tesla V100 32G GPUs. Training the large and base models takes 75 and 25 hours, respectively.

We develop two kinds of approaches to analyze their generalization abilities, namely Auto-GUI_{unified} and Auto-GUI_{separate}. Specifically, Auto-GUI_{separate} is trained and evaluated independently on each subset. Auto-GUI_{unified} is a unified model trained on the training sets of each subset and evaluated on each test set. As the GoogleApps subset is 10-100 times larger than the other subsets, using all the training data to train a unified model would suffer from the data imbalance issue (Zhang et al., 2022). Therefore, we only use 10% training data of GoogleApps. At the same time, the overall computation cost can also be saved by 80%. We use Auto-GUI_{unified} as the default model for analysis unless otherwise stated.

4.5 Main Results

Table 2 shows the main results. Based on the results, we have the following observations.

(i) Auto-GUI_{unified} achieves the best overall performance compared with all the baselines. Compared with separate (not unified) models, Auto-GUI_{unified} shows general effectiveness across various tasks. The results show that a unified multimodal model out of first principles thinking can

<table>
<thead>
<tr>
<th>Model</th>
<th>Unified w/o Anno.</th>
<th>Overall</th>
<th>General Install GoogleApps Single WebShopping</th>
</tr>
</thead>
<tbody>
<tr>
<td>PaLM 2-CoT</td>
<td>✓</td>
<td>39.6</td>
<td>-</td>
</tr>
<tr>
<td>ChatGPT-CoT</td>
<td>✓</td>
<td>7.72</td>
<td>5.93 4.38 10.47 9.39 8.42</td>
</tr>
<tr>
<td>GPT-4V</td>
<td>✓</td>
<td>52.96</td>
<td>43.01 46.14 49.18 78.29 48.18</td>
</tr>
</tbody>
</table>

Fine-tuned Llama 2    | ✗      | 28.40   | 28.56 35.18 30.99 27.35 19.92               |

BC-single            | ✗      | 68.7    | -                                            |
BC-history           | ✗      | 73.1    | 63.7 77.5 75.7 80.3 68.5                    |

Auto-GUI<sub>separate</sub> | ✗      | ✓       | 74.07 65.94 77.62 76.45 81.39 69.72         |
Auto-GUI<sub>unified</sub> | ✓      | ✓       | 74.27 68.24 76.89 71.37 84.58 70.26         |

Table 2: Main results (%). Segment 1: in-context learning LLM baselines; Segment 2: fine-tuned Llama 2 baseline; Segment 3: specialized agent baselines; Segment 4: our Auto-GUI results. Prior published best results are marked with an underline. “Unified” means a general model that can work across subsets. “w/o Anno.” means no screen description is needed. The PaLM-CoT and BC results are from Rawles et al. (2023). The GPT-4V result is from Yan et al. (2023). The other results are based on our own implementations. The overall score is computed as the average accuracy on all the subsets. The best average result is in bold face.

![Figure 3](image_url)

Figure 3: Performance of Auto-GUI with respect to varying length of chain of actions.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-GUI</td>
<td>74.27</td>
</tr>
<tr>
<td>w/o future action plan</td>
<td>73.78</td>
</tr>
<tr>
<td>w/o previous action history</td>
<td>68.81</td>
</tr>
<tr>
<td>w/o chain of actions</td>
<td>68.53</td>
</tr>
<tr>
<td>w/o coordinate normalization</td>
<td>70.23</td>
</tr>
</tbody>
</table>

Table 3: Ablation study of Auto-GUI.

serve as a strong autonomous agent. Compared with previous BC models, Auto-GUI<sub>unified</sub> has two major advantages. First, Auto-GUI<sub>unified</sub> is a unified model that can be adapted to different scenarios without the need to train specific models for each task. Second, Auto-GUI<sub>unified</sub> does not need additional annotations and is more practical in real-world applications. Furthermore, Auto-GUI yields divergent performance across subsets. We provide the explanation in Appendix C.1 to save space.

(ii) Both the chain of actions and coordinate normalization contribute to the overall performance (+5.74% and 4.04%, respectively), as evidenced by the ablation study in Table 3. Additionally, we set the maximum numbers of the previous actions and future actions to 8 and 4, respectively. The choice is made according to our analysis of the General subset with Auto-GUI<sub>separate</sub> (Figure 3). The model under those setups achieves the optimal performance, and neither the input nor output sequence lengths exceed the model limit.

(iii) For the LLMs, using either prompting or fine-tuning techniques does not achieve competitive performance compared with multimodal approaches. The most plausible reason is that they learn from the parsed HTML elements of the screen so that they may suffer from information loss compared with more informative vision features of the screens. Specifically, we find that ChatGPT is quite accurate at predicting the action type but fails at lower-level executions (Appendix 5.2).

5 Analysis

5.1 Analysis of Auto-GUI by Category

To dive into the capability of Auto-GUI, we calculate the click region accuracy, scroll direction accuracy, action type accuracy, and typed text accuracy. Figure 4 presents the results. We see that Auto-GUI achieves over 90% action type accuracy on average. In contrast, the major challenges lie within the click region and scroll direction predictions. Although the model is able to predict the right action most of the time, it tends to click a
5.2 Comparison with ICL by Category

To understand how the ICL baseline performs on our task and assess the advantage of Auto-GUI, we conduct a category comparison with ChatGPT. In Table 6, we see that the ICL method (ChatGPT) is quite accurate at predicting the action type (41.72%) but fails at lower-level executions, e.g., clicking positions (8.5%) and scrolling directions (4.0%). The results show that using HTML-based layout information is not enough to accurately execute actions. In contrast, Auto-GUI has the advantage of predicting both action types and performing low-level executions by leveraging multimodal perception and the chain-of-action technique.

5.3 Generalization Ability

As our approach is designed under first principles thinking and does not rely on pre-defined internal APIs, it could be easily generalized to new task domains. To verify the generality, we evaluate the performance of Auto-GUIseparate on each subset in Figure 5. For example, we train an Auto-GUIseparate model on the training set of General and then test its performance on the tests of each subset.

We see that our approach is able to achieve a decent performance, though the domains vary. This result reveals that the model could capture general knowledge for the GUI control task; thus is applicable to different domains. In addition, the unified model Auto-GUIunified can serve as a potential choice in real-world applications owing to more coverage of training data.

5.4 Comprehensive Analysis

Here, we present a comprehensive analysis of the choice of pre-trained features and model scale. The results are summarized in Table 4.

(i) Pre-trained Features. There are two kinds of pre-trained features used in this work, the vision features and language model weights. For vision features, we compare two popular types, CLIP (Radford et al., 2021) and BLIP-2 (Li et al., 2023). We observe that BLIP-2 achieves relatively better performance. Therefore, we use BLIP-2 by default in Auto-GUI. For pre-trained language model weights, we compare initializing the model with the vanilla T5 (Raffel et al., 2020), FLAN-T5 (Chung et al., 2022), and FLAN-Alpaca (Taori et al., 2023a) weights under the large size. We see that FLAN-Alpaca achieves the best performance as it has been optimized with Stanford Alpaca synthetic instruction tuning data.

(ii) Model Scale. Compared with the performance gains from our technique components (chain of actions and coordinate normalization) in Table 3, the benefit of scaling parameter size becomes relatively marginal. As we observe that a larger model size does not lead to dramatic improvement in performance, we do not scale the model scale but
TABLE 4: Results varying vision features and pre-trained language model weights.

<table>
<thead>
<tr>
<th>Model</th>
<th>Overall</th>
<th>General</th>
<th>Install</th>
<th>GoogleApps</th>
<th>Single</th>
<th>WebShopping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-GUI on CLIP</td>
<td>71.84</td>
<td>66.28</td>
<td>74.40</td>
<td>69.71</td>
<td>81.60</td>
<td>67.23</td>
</tr>
<tr>
<td>Auto-GUI on BLIP-2</td>
<td>74.27</td>
<td>68.24</td>
<td>76.89</td>
<td>71.37</td>
<td>84.58</td>
<td>70.26</td>
</tr>
<tr>
<td>Auto-GUI on Vanilla-T5\textsubscript{large}</td>
<td>72.98</td>
<td>66.61</td>
<td>75.40</td>
<td>70.86</td>
<td>83.47</td>
<td>68.54</td>
</tr>
<tr>
<td>Auto-GUI on FLAN-T5\textsubscript{large}</td>
<td>73.36</td>
<td>67.59</td>
<td>76.35</td>
<td>70.71</td>
<td>83.01</td>
<td>69.12</td>
</tr>
<tr>
<td>Auto-GUI on FLAN-Alpaca\textsubscript{large}</td>
<td>74.27</td>
<td>68.24</td>
<td>76.89</td>
<td>71.37</td>
<td>84.58</td>
<td>70.26</td>
</tr>
<tr>
<td>Auto-GUI on FLAN-Alpaca\textsubscript{base}</td>
<td>71.38</td>
<td>65.26</td>
<td>74.90</td>
<td>68.70</td>
<td>81.20</td>
<td>66.83</td>
</tr>
<tr>
<td>Auto-GUI on FLAN-Alpaca\textsubscript{large}</td>
<td>72.84</td>
<td>66.97</td>
<td>75.93</td>
<td>70.29</td>
<td>82.56</td>
<td>68.46</td>
</tr>
<tr>
<td>Auto-GUI on FLAN-Alpaca\textsubscript{large}</td>
<td>74.27</td>
<td>68.24</td>
<td>76.89</td>
<td>71.37</td>
<td>84.58</td>
<td>70.26</td>
</tr>
</tbody>
</table>

Table 5: Computation cost of Auto-GUI and Llama. “s/n” is computed by time (s) divided by the number of inferences (n). Llama 2 is hosted with 8-bit quantization and float16 precision to improve the inference speed.

<table>
<thead>
<tr>
<th>Model</th>
<th>Feature Extraction (s/n)</th>
<th>Model Inference (s/n)</th>
<th>Peak GPU Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-GUI\textsubscript{base}</td>
<td>0.06</td>
<td>0.19 (45x)</td>
<td>4.6 (10x)</td>
</tr>
<tr>
<td>Auto-GUI\textsubscript{large}</td>
<td>0.06</td>
<td>0.59 (15x)</td>
<td>8.2 (6x)</td>
</tr>
</tbody>
</table>

Llama 2 - - 8.5 49.7

Table 6: Category comparison with the ICL baseline on the General test set.

<table>
<thead>
<tr>
<th>Model</th>
<th>Overall</th>
<th>Action Type</th>
<th>Click</th>
<th>Scroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChatGPT</td>
<td>5.93</td>
<td>41.72</td>
<td>8.50</td>
<td>4.00</td>
</tr>
<tr>
<td>Auto-GUI</td>
<td>68.24</td>
<td>87.03</td>
<td>58.34</td>
<td>82.74</td>
</tr>
</tbody>
</table>

GUI environment without environment parsing or application-dependent API access. In addition, we propose a chain-of-action technique that leverages the previously executed actions and future action plans to help the agent decide what action to execute. Experimental results show that Auto-GUI achieves superior performance to previous prompting-based and fine-tuning baselines. We show that it is possible to achieve state-of-the-art performance by an end-to-end model without relying on external tools and application-specific APIs to parse the environment and interpret the predicted actions. Besides the strong performance and generality across domains, Auto-GUI is able to infer an action in less than one second.

**Limitations**

We acknowledge two primary limitations in our study. First, we opted not to extend the approach to extremely large models because our work aims to provide a simple yet effective solution for GUI agents. Our findings suggest that scaling may not be fundamentally advantageous in GUI problems. The significance of model scale tends to diminish—increasing the model size does not necessarily result in a substantial performance enhancement. Second, our experiments and analysis were exclusively conducted on AITW, which is the largest-scale and widely recognized benchmark dataset in the research line of autonomous GUI agents, to provide timely and pertinent insights. Given the rapid development of the field, we anticipate future studies to explore the application of our approach on other benchmark datasets as they become available.


A Data Details

A.1 Data Examples

We show the data examples from the AITW benchmark dataset (Rawles et al., 2023). Figures 6-9 show the examples in each subset, i.e., General, Install, GoogleApps, Single, and WebShopping. The gold actions for each screen are depicted in the illustrations for reference.

A.2 Data Statistics

We use the AITW benchmark dataset (Rawles et al., 2023). AITW is a large-scale benchmark dataset for GUI control, which contains natural language instructions, screenshots, and actions. There are 715K episodes spanning 30K unique instructions, covering diverse multi-step tasks such as application operation, web searching, and web shopping, on over 350 Apps and websites. This dataset covers various device types and operation systems in varying screen resolutions to ensure generality. There are five subsets in the benchmark dataset: General, Install, GoogleApps, Single, and WebShopping.

(i) General contains miscellaneous tasks that need interaction with third-party Apps and websites, as well as question answering.

(ii) Install contains tasks related to installing, uninstalling, logging Apps, and App login support.

(iii) GoogleApps contains tasks about manipulating various Google applications such as Gmail, Calendar, Photos, and Settings.

(iv) Single contains atomic tasks (e.g., “upvote the post”) whose preceding actions have been already completed (e.g., opening Instagram, going to home feed, looking at a post).

(v) WebShopping contains tasks related to online shopping on E-commerce websites, e.g., searching for an item, adding an item to the cart, and viewing the shopping cart.

Table 7 presents the data statistics of the AITW dataset. Each subset is split episode-wise into a training, validation, and test set (80/10/10%).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Episodes</th>
<th>Screens</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>9,476</td>
<td>85,413</td>
<td>545</td>
</tr>
<tr>
<td>Install</td>
<td>25,760</td>
<td>250,058</td>
<td>688</td>
</tr>
<tr>
<td>GoogleApps</td>
<td>625,542</td>
<td>4,903,601</td>
<td>306</td>
</tr>
<tr>
<td>Single</td>
<td>26,303</td>
<td>85,668</td>
<td>15,366</td>
</tr>
<tr>
<td>WebShopping</td>
<td>28,061</td>
<td>365,253</td>
<td>13,473</td>
</tr>
</tbody>
</table>

Table 7: Dataset statistics.

B Implementation Details

B.1 Coordinate Normalization

Recall that a target action consists of four components: action type, touch point, lift point, and typed text. We consider six action types: dual-point gesture, type, go_back, go_home, enter, and status_complete. A dual-point gesture comprises a touch point and a lift point with \([y, x]\) coordinates. The gesture actions ensure a flexible action space and can represent clicks and scrolls at arbitrary locations. For example, a gesture action \{"touch_point": [0.7761, 0.7089], “lift_point”: [0.7761, 0.7089]\} means clicking at the coordinate [0.7761, 0.7089], while a gesture action \{"touch_point": [0.1898, 0.4477], “lift_point”: [0.8242, 0.4077]\} means scrolling down. A type action means typing a text and the text is placed in the <typed_text> field. The other ac-


<table>
<thead>
<tr>
<th>Action Type</th>
<th>Target Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>dual-point gesture (click)</td>
<td>“action_type”: 4, “touch_point”: [0.8497, 0.5964], “lift_point”: [0.8497, 0.5964], “typed_text”: “”</td>
</tr>
<tr>
<td>dual-point gesture (scroll)</td>
<td>“action_type”: 4, “touch_point”: [0.2, 0.5], “lift_point”: [0.8, 0.5], “typed_text”: “”</td>
</tr>
<tr>
<td>type</td>
<td>“action_type”: 3, “touch_point”: [-1.0, -1.0], “lift_point”: [-1.0, -1.0], “typed_text”: “what’s the news in chile?”</td>
</tr>
<tr>
<td>go_back</td>
<td>“action_type”: 5, “touch_point”: [-1.0, -1.0], “lift_point”: [-1.0, -1.0], “typed_text”: “”</td>
</tr>
<tr>
<td>go_home</td>
<td>“action_type”: 6, “touch_point”: [-1.0, -1.0], “lift_point”: [-1.0, -1.0], “typed_text”: “”</td>
</tr>
<tr>
<td>enter</td>
<td>“action_type”: 7, “touch_point”: [-1.0, -1.0], “lift_point”: [-1.0, -1.0], “typed_text”: “”</td>
</tr>
<tr>
<td>status_complete</td>
<td>“action_type”: 10, “touch_point”: [-1.0, -1.0], “lift_point”: [-1.0, -1.0], “typed_text”: “”</td>
</tr>
</tbody>
</table>

Table 8: Target output examples after the coordinate normalization.

We observe that high-precision coordinates are not necessary for representing a click or scroll action. Therefore, we apply normalized values of the coordinates, which helps accelerate convergence and mitigate the ambiguity of coordinates. The normalization is applied to click and scroll actions. For click actions, we keep four decimal places. For scroll actions, we first determine the scroll direction with the touch and lift points. Then, we transform the touch and lift points into fixed directional coordinates as follows: “up”: {[0.8, 0.5], [0.2, 0.5]}, “down”: {[0.2, 0.5], [0.8, 0.5]}, “left”: {[0.5, 0.8], [0.5, 0.2]}, “right”: {[0.5, 0.2], [0.5, 0.8]}, where 

{[·, ·]} consists of the touch point and lift point in the first [·] and second [·]. We provide examples of target actions in Table 8.

Figure 8: An example episode from GoogleApps.

Figure 9: An example episode from WebShopping.

B.2 Baselines

We adopt three types of baselines for comparisons. The baselines encompass the in-context learning (ICL) and fine-tuning paradigms, along with various backbone models of different sizes. This choice of baselines allows for a comprehensive comparison with our proposed approach.

(i) In-context Learning LLMs. Few-shot PaLM 2, ChatGPT (turbo-3.5) are adopted. Following previous studies (Rawles et al., 2023; Wang et al., 2023a), we feed the LLM a textual description of the screen and a user instruction. The textual description of the screen is formatted as an HTML

3145
Table 9: Ablation study of Auto-GUI design components. We adopt Auto-GUI unified for analysis.

<table>
<thead>
<tr>
<th>Model</th>
<th>Overall</th>
<th>General</th>
<th>Install</th>
<th>GoogleApps</th>
<th>Single</th>
<th>WebShopping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-GUI</td>
<td>74.27</td>
<td>68.24</td>
<td>76.89</td>
<td>71.37</td>
<td>84.58</td>
<td>70.26</td>
</tr>
<tr>
<td>w/o previous action history</td>
<td>73.78</td>
<td>67.97</td>
<td>76.66</td>
<td>71.00</td>
<td>83.64</td>
<td>69.62</td>
</tr>
<tr>
<td>w/o future action plan</td>
<td>68.81</td>
<td>59.01</td>
<td>72.34</td>
<td>67.95</td>
<td>81.53</td>
<td>63.24</td>
</tr>
<tr>
<td>w/o chain of actions</td>
<td>68.53</td>
<td>58.99</td>
<td>72.06</td>
<td>67.50</td>
<td>81.25</td>
<td>62.86</td>
</tr>
<tr>
<td>w/o coordinate normalization</td>
<td>70.23</td>
<td>63.79</td>
<td>73.28</td>
<td>66.63</td>
<td>82.11</td>
<td>65.33</td>
</tr>
</tbody>
</table>

Table 10: Results of Auto-GUI when using annotated screen descriptions.

<table>
<thead>
<tr>
<th>Model</th>
<th>Overall</th>
<th>General</th>
<th>Install</th>
<th>GoogleApps</th>
<th>Single</th>
<th>WebShopping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-GUI base</td>
<td>72.84</td>
<td>66.97</td>
<td>75.93</td>
<td>70.29</td>
<td>82.56</td>
<td>68.46</td>
</tr>
<tr>
<td>w/ Screen Descriptions</td>
<td>75.54</td>
<td>70.30</td>
<td>78.05</td>
<td>73.04</td>
<td>85.31</td>
<td>71.00</td>
</tr>
</tbody>
</table>

syntax, providing the information of GUI elements derived from OCR detection and icon detection from external tools (Rawles et al., 2023). The model is required to predict an action among predefined actions. If the action is clicking, the model will be required to provide the index of the clicked GUI element. Alternatively, the model needs to provide the scroll direction if the action is scrolling. In addition, 5-shot CoT prompting is leveraged to improve the performance (Appendix B.3). In addition, we report the results of the multimodal GPT-4V by taking the vision image and action history as the input based on Yan et al. (2023).

(ii) Fine-tuned LLMs. We adopt Llama 2 (Touvron et al., 2023) as the baseline and fine-tune it with LoRA. We feed the model with the user instruction and the screen descriptions in HTML syntax (the same as adopted for in-context learning LLMs). The model is expected to predict the action in the same output format as in-context learning LLMs. As fine-tuning an LLM is expensive, we randomly sample 1% training data to help the LLM adapt to our tasks.

(iii) Specialized GUI Agent. We adopted the Behavioural Cloning (BC) agent, which reported the state-of-the-art performance in Rawles et al. (2023). BC is a Transformer-based architecture that takes a task instruction, the current screen, and a stacked history of screen observations and actions as input. The task instruction and OCR-detected texts are encoded by a pre-trained BERT. The icons are represented by the embeddings for each of the bounding box points. The screen history is modeled by the \{x, y\} positions of the touch and lift actions. All the embedded representations are fused to predict the action by a decoder. There are two BC variants, BC-single and BC-history, depending on whether the model takes as input the screen-action history.

B.3 LLM Prompt
We use the prompt in Figures 10-11 for PaLM 2-CoT and ChatGPT-CoT owing to its optimal performance reported in Rawles et al. (2023).

C Further Analysis
C.1 Subset Analysis
We notice that Auto-GUI unified performs relatively inferior to BC-history on the two App-centered subsets, Install and GoogleApps. It is reasonable because we only use 10% training data of GoogleApps considering the data balance and computation overhead. We observe that the performance does not improve when we use all the training data of GoogleApps, possibly due to the data imbalance issue (Zhang et al., 2022). In contrast, our separate model Auto-GUI separate can achieve better performance than BC-history, showing that our approach is better than BC-history under the same training setting. As we aim to study a simple and unified approach that achieves generally strong performance, we leave the treatment of the data imbalance issue in future work.

C.2 Ablation Study
Table 9 shows the detailed results of the ablation study. We see that both the chain of actions and coordinate normalization contribute to the overall performance (+5.74% and 4.04%, respectively).

C.3 Using Screen Descriptions
We are interested in whether Auto-GUI can be further improved when screen annotations are available. Therefore, we incorporate screen descriptions
containing icon and text information, organized in HTML syntax, into our language input $X_{\text{language}}$. Detailed examples of screen descriptions can be found in the “Screen” block in Appendix B.3.

In Table 10, we see that Auto-GUI can perform better when the annotated screen descriptions are available. The results show that there is still room for performance gains for Auto-GUI. However, as the annotations are not always available in real-world applications, we do not include them by default in our framework.
Given a mobile screen and a question, provide the action based on the screen information.

**Available Actions:**
- action_type: click, idx: <element_id>
- action_type: type, text: <text>
- action_type: navigate_home
- action_type: navigate_back
- action_type: scroll, direction: up
- action_type: scroll, direction: down
- action_type: scroll, direction: left
- action_type: scroll, direction: right

**Previous Actions:**
- step_idx: 0, action_description: press [HOME key]
- step_idx: 2, action_description: click [Google Icon]
- step_idx: 3, action_description: click [search for hotels]

**Screen:**
- p id=0 class="IconGoogle" alt="Google Icon">
- p id=2 class="text" alt="search for hotels">
- p id=3 class="text" alt="in">
- p id=4 class="text" alt="mexico city mexico">
- p id=5 class="IconMagnifyingGlass" alt="Search Icon">
- p id=7 class="text" alt="Select all">
- p id=8 class="text" alt="Cut">
- p id=9 class="text" alt="Copy">
- p id=10 class="text" alt="hotel in mex">
- p id=11 class="IconMagnifyingGlass" alt="Search Icon">
- p id=12 class="text" alt="best hotel">
- p id=13 class="text" alt="mexico city mexico">
- p id=14 class="text" alt="in">
- p id=15 class="IconMagnifyingGlass" alt="Search Icon">
- p id=16 class="text" alt="K">
- p id=17 class="text" alt="hotel ciudad">
- p id=18 class="text" alt="de mexico">
- p id=19 class="text" alt="gran">
- p id=20 class="IconVBackward" alt="Left Icon">
- p id=21 class="IconNavBarCircle" alt="Home Icon">
- p id=22 class="IconNavBarRect" alt="Overview Icon">

**Instruction:** What time is it in Berlin?
**Answer:** Let’s think step by step. I see unrelated search results in the Google app, I must clear the search bar, so the action is {"action_type": "click", "idx": 1}

**Previous Actions:**
- step_idx: 0, action_description: click [DISMISS]
All the features you love with a fresh look.

GOT IT

Instruction: open app "Google Play services"
Answer: Let’s think step by step. I see the GMail app, I need to open the app drawer, so the action is

Previous Actions:
Screen:

Instruction: open app "Messenger Lite" (install if not already installed)
Answer: Let’s think step by step. I see the home screen, I need to open the app drawer, I should swipe up, so the action is

Previous Actions:

Screen:

Instruction: Search for hotels in Chicago.
Answer: Let’s think step by step. I see the app drawer, I need to search, so the action is

Previous Actions:

Screen:

Figure 11: LLM Prompt (Part-II).